Mobile Python

Rapid Prototyping of Applications on the Mobile Platform

Jürgen Scheible and Ville Tuulos

Reviewed by

Panos Asproulis, Mal Minhas, Tim Ocock, Mark Shackman, Ian Weston

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb

BICENTENNIAL 1807 WILEY 2007 BICENTENNIAL

John Wiley & Sons, Ltd

Copyright © 2007 John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England

Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit our Home Page on www.wileyeurope.com or www.wiley.com

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 6045 Freemont Blvd, Mississauga, Ontario, L5R 4J3, Canada

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Anniversary Logo Design: Richard J. Pacifico

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN: 978-0-470-51505-1

Typeset in 10/12pt Optima by Laserwords Private Limited, Chennai, India
Printed and bound in Great Britain by Bell & Bain, Glasgow
This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

Contents

Contributors

Head of Symbian Press

Freddie Gjertsen

Authors

Jürgen Scheible
Ville Tuulos

Symbian Press Editorial

Managing Editor

Satu McNabb

Reviewers and Additional Contributors

Panos Asproulis
Jukka Laurila
Joe McCarthy
Timo Ojala
Mark Shackman

About the Authors

Jürgen Scheible

Jürgen Scheible is a designer, media artist and musician who holds a degree in telecommunications from Karlsruhe, Germany. After graduating, he worked for eight years at Nokia in Finland pursuing various positions such as programmer, product manager and competence transfer manager. Besides his occupation, he performed and produced music as well as media art under the pseudonym Lenin's Godson.

In 2003, he left his engineering career to concentrate full-time on his creative career, because he felt his heart was much more in his artistic works than in engineering. In 2004, he became a doctoral student at the Media Lab at the University of Art and Design, Helsinki, where he established the Mobile Hub, a prototype development environment for mobile client and server applications. It has a strong focus on artistic approaches and creative design, and serves as a resource to art and design students who use mobile technology as part of their projects. His doctoral research focuses on designing multimodal user interfaces for creating and sharing interactive artistic experiences.

Since 2004, he has been evangelizing Python for S60 as one of its pioneers. He is internationally active having given talks and taught innovation workshops in both academic and professional settings on more than 40 occasions, in places such as Stanford University, MIT, NTU Taiwan, Yahoo Research Berkeley, Tsinghua University Beijing, Nokia and Nokia Siemens Networks, in more than 17 countries. His focus is on rapid mobile application prototyping using creative approaches for innovation.

In 2006, he spent several months as a visiting scientist at MIT, Boston in the Computer Science and Artificial Intelligence Laboratory (CSAIL).

Jürgen was recognized as a Forum Nokia Champion in 2006 and 2007 for his driving vision of building bridges between art, engineering and research. He was one of the winners of the ACM Computers in Entertainment Scholarship Award in 2006 and of the Best Arts Paper Award at ACM Multimedia 2005 conference.

The philosophy behind his works is to bring back the depth of human feelings and emotional aspects to the digital world which, in his opinion, were lost with the arrival of the fast-paced digital production technology. By inspiring others with his works, he gets inspired himself. This leads him to many new ideas for designing new kinds of interactive experiences for people, especially in the area of mobile phone applications that fuse the real and the virtual worlds. He believes this era will change the way we live and communicate in the future and it will transform societies. Therefore it is important, in his opinion, to design for these coming applications.

Ville Tuulos

Ville Tuulos is currently a researcher in the Department of Computer Science, University of Helsinki, Finland. He has more than 15 years' experience of creative hacking, including data visualization, web search engines, and machine-learning algorithms. He has been an enthusiastic Pythonista since 2000 and he has been exploring and extending the limits of Python for S60 since 2005. He has used it to implement, among others, real-time image processing algorithms, various positioning techniques and an urban game for 200 players in New York City.

Authors' Acknowledgments

We'd like to express our gratitude to all the people who played a part in developing this book. First, we'd like to thank the editors that worked on this project: Satu McNabb at Symbian; Hannah Clement, Andrew Kennerley and Rosie Kemp at Wiley; and Shena Deuchars at Mitcham Editorial. We'd also like to thank Wiley and Symbian, in general, for supporting this book project. We are glad to be part of it.

Thanks also to everyone who took part in the early review of this book: Joe McCarthy, Panos Asproulis, Mark Shackman, Tim Ocock, Timo Ojala and Jukka Laurila, who also contributed Appendix A.

And for creating such an enjoyable and useful thing as Python for S60, we thank Jukka Laurila, the mastermind behind Python for S60 at Nokia, as well as Erik Smartt and Kari Pulli who greatly supported its bringing it to life. We owe an especially large debt to Guido van Rossum, the creator of Python, for this beautiful and fun language, and the growing Python for S60 community for their engagement and contributions to the mobile space.

We are grateful also to Joe McCarthy for the fruitful discussions we had and writing an executive book summary for us. A big thanks goes to Tomi Silander for his experiments, source code and information on robotics using Roombas and Python for S60.

Finally, we want to thank Harri Pennanen, for helping to spread the knowledge about Python for S60 to universities around the globe and his great managerial support for realizing Manhattan Story Mashup.

Jürgen's Acknowledgments

I'd like to thank David Wood and Symbian for giving me a chance to work on this book project, being able to share my vision of user-driven

innovation in the mobile space with a larger audience and to make them aware of this great toolkit named Python for S60. This platform can bring so much fun and inspiration to one's own and other people's lives. It's been so much fun to write this book.

I am grateful to Professor Timo Ojala for his relentless support during my studies and research in which Python for S60 has played a crucial role from the beginning. I also want to thank Professor Philip Dean for helping me to set up the Mobile Hub at the Media Lab of University of Art and Design, Helsinki.

My special thanks go to Joost Bonsen, my great mentor during my time at MIT, who strongly encouraged and supported me to go forward with this book and writing the initial proposal.

Also I want thank Eric von Hippel. At MIT, it was great to get to know him and his inspiring arguments about user-driven innovation which I have put at the heart of my evangelizing approach to Python for S60 and this book.

I'd like to thank all the students who have participated in my workshops and tutorials. Your feedback, needs and inspiration have played a huge role in shaping my contributions to this book.

Finally a few personal notes of thanks: to Steffi, for being patient, understanding and always supportive during the entire time of writing, thinking and travelling; and to my late mom, for helping me finding my own sources of inspiration, motivation and depth in life.

Ville's Acknowledgments

Since 2000, I have been using Python on the server side for rapid prototyping and as a glue language to handle large-scale scientific computation. As an enthusiastic Linux user, I have become accustomed to systems that are open, pragmatic and 'shiny, beautiful things that you can poke at and bend to your will', to quote the head-honcho of Python for S60, Jukka Laurila. Thanks to his sense of aesthetics, Python for S60 has become one of those beautiful things – and shiny too!

Python for S60 may bring the mobile platform closer to the Internet with respect to open innovation, mad experimentation and community-oriented development. Jürgen is doing great work on evangelizing this idea, so I was delighted to accept his invitation to join him in writing this book and take the opportunity to share my knowledge about Python for S60 with the readers.

Since 2005 I have implemented real-time image processing algorithms, a framework for self-migrating code (with Jukka Perkiö), Bluetooth and GSM-based positioning systems and a large-scale urban game, among others, using Python for S60. I want to thank Henry Tirri from Nokia

Research Center for motivating many of these intellectual challenges and for his 'Just do it' attitude, which fits Python for S60 so perfectly.

Also, I want to thank Professor Petri Myllymäki and all my colleagues in the Complex Systems Computation research group at the University of Helsinki, for their no-nonsense, Zen-like insights into various technologies, including Python for S60.

Last but not least, a huge hug to my wife Heli Tuulos for keeping me updated with the latest web techniques and for tolerating my endless stream of questions regarding details of the book.

Symbian Press Acknowledgments

Symbian Press wishes to thank the authors, Jürgen Scheible and Ville Tuulos, for giving us the opportunity to publish this unique book and for working so hard during the past year. Without your efforts, publishing this book would not have been possible.

We'd also like to acknowledge reviewers Ian Weston, Mal Minhas, Mark Shackman, Panos Asproulis and Tim Ocock for giving their time and sharing their technical knowledge. Finally, we'd like to thank Phil Northam who first discussed the book with Jürgen and got the ball rolling!

Forewords

Guido van Rossum

While skimming the manuscript of this book, I couldn't help wanting to whip out my phone and start hacking it right there and then – and it wasn't just because I was procrastinating on writing this foreword.

As Python's creator, I'm proud that this book uses Python, but hardly surprised. Python is simply one of the best languages for exploratory programming, which is what this book is all about. Python is perfect for the do-it-yourself experiments, prototypes and games that pop off the pages everywhere in this book.

Python has been ported to many platforms, but the port to Nokia's S60 system is in many ways unique: It is by far the smallest platform to which Python has successfully been ported, and it has potentially the largest number of users. It is also one of the most *connected* platforms: camera, phone, Bluetooth, Internet and more. All this makes for a very exciting platform and I'm glad that Jürgen and Ville have written this book showing everyone how easy it is to program your own phone!

As will become clear when you read through the many examples in the book, programming a phone these days doesn't require a degree in wireless communication. I predict that even people who haven't written a single computer program in their life will be able to follow the instructions in this book, and soon will be writing their own programs to hook up their phone to the rest of their life in interesting ways.

While this book isn't a tutorial for the Python programming language, all the concepts necessary for understanding the example programs are explained clearly, so you'll be at least an apprentice-level Python

programmer by the time you've finished the book. At which point you may want to further test and improve your skills by writing code for a PC or Mac. The free online tutorials available from ***http://python.org*** will be at your service then.

Programming can be fun, and using Python is one of the best ways to have fun programming, for your phone or for any other computer!

Enjoy,

Guido van Rossum

Mountain View, CA

July 2007

Eric von Hippel

I am very happy to write a foreword to this wonderful book. In it, Jürgen Scheible and Ville Tuulos teach us how to simply and quickly use a Python-based toolkit to create custom applications for mobile device platforms, such as mobile phones.

Jürgen and Ville have long been on a mission to open up mobile devices to the many millions of us who really want to create our own applications for these devices – but who do not have the specialized technical skills that have been needed until now. Today, the situation facing users of mobile platforms is very much like that faced by music fans a few years ago. Many music fans wanted to create and modify music using digital tools, but few had the programming skills needed to do so. Then along came simple, user-friendly and very capable software, such as Propellerhead Software music toolkits, and many more of us were suddenly empowered to create our own music using the wonderful possibilities opened up by powerful digital tools.

In this book, Jürgen and Ville help create a similar revolution in the field of mobile devices. Using the simple but powerful kit of tools that they teach us, we can quickly learn how to create and insert custom, Python-based programs into 'open' mobile phones and other devices. Jürgen and Ville have themselves used this toolkit to build many useful prototypes and applications. In addition, they have learned effective ways to teach us to accomplish similar things. They have taught these capabilities to many groups, small and large. As a result, this book is very user-friendly and also very effective. There are many examples and many trial scripts for us to create and immediately apply in order to learn by doing.

Thank you so much for all your hard work and for giving us this intellectual gift, Jürgen and Ville!

Eric von Hippel

Cambridge, MA, USA

June 2007

1

Introduction and Basics

This practical hands-on book introduces the Python programming language for rapid prototyping of mobile device applications. It effectively teaches how to program easily on Nokia smartphones that are based on Symbian OS and the S60 platform. A wide range of smartphone functionalities are covered, including camera, sound, graphics, Bluetooth, Internet, positioning, SMS messaging and many more.

Mobile Python – or, more formally, Python for S60 (see Figure 1.1) – empowers you to do fun and engaging stuff with your mobile phone. You can start programming shortly after getting into this book. Being able to see results quickly on the phone guarantees to bring inspiration and makes programming these gadgets fun!



Figure 1.1 Python for S60

Development on the Symbian platform has been time-consuming in the past and it has required in-depth knowledge of C++ or Java. Python for S60 remedies this problem. It is easy to learn and takes only a few days to get into most of its features. Novice programmers, artists and people from creative communities can innovate and contribute applications to the mobile space.

Python for S60 brings the increasingly popular Python programming language to the mobile platform. You can use this book to learn the Python programming language by way of Python for S60 or use your previous Python knowledge to get into mobile programming in no time. Similar to traditional Python, Python for S60 is released under an open-source license, so you will be backed up by an enthusiastic community of talented developers and a large library of extension modules.

Python for S60 allows you to go through a fast iterative design cycle by providing an elegantly simple but powerful platform for your programs, which can be rapidly adapted to real-world requirements. With completely free and open tools you are able to create useful and appealing applications based on your own ideas.

Our message to all you creative and innovative people out there is: use your talent, skills, ideas and energy to inspire the world! May this book help you to do so!

Although the book is written in a style that beginners can cope with, it hopefully also offers experienced programmers new insights and in-depth knowledge. It is not a traditional programming text book with meticulous coverage of every aspect of a programming language. Instead it adopts a light and engaging tone that helps the reader to proceed through the chapters in a practical hands-on manner, steadily increasing knowledge through learning by doing.

The material in this book has been reformulated and refined through dozens of workshops and tutorials offered by the authors at a wide variety of institutions and companies across 17 countries, including MIT, Stanford University, the University of Art and Design, Helsinki, and Yahoo Research, Berkeley. Among the hundreds of applications created by participants in these events are games for social interaction; applications for interacting with large public displays, sensors or robots; and personal organizer applications. This book is intended to broaden the audience for these kinds of applications even further, enabling you to create new mobile applications that may have seemed to be out of your, or anyone's, reach before.

We have used the book's material to build many applications and prototypes for companies such as Nokia. We have also helped several universities around the globe to conduct research projects on robotics, sensor networks, positioning and data collection using Python for S60.

Mobile phones are carried by over two billion people – far more than the 200 million who carry laptop computers. Despite the greater penetration of mobile phone *users*, the number of mobile phone *programmers* is far lower than the number of people who have learned to program personal computers. This book aims to change that.

There is no doubt that the built-in features of mobile phones have empowered a generation of people to connect with others more effectively than ever before. However, these standard applications are just scratching the surface with respect to unleashing the true creative potential of our culture. We can expect to see a user-driven innovation era and user-generated mobile applications are in the near reach. This book is inspired by [von Hippel 2005] and lays out a practical path for how innovation driven by lead users can become a reality on the mobile platform.

By reading this book, you will become fully equipped to be one of the new lead users. You may gain inspiration and motivation that turns you into an innovator and a contributor to the developer community for mobile applications.

1.1 Why Does Python Make a Difference?

Only skilled and experienced programmers were previously able to build mobile applications using C++ or Java. As a result, many people often gave up early or never really started.

The emergence of Python for S60 offers a crucial turning point, as it brings the Python programming language to the mobile space. This makes mobile development approachable for many new developers that were previously excluded.

Python for S60 can drastically reduce development time; it allows development with completely free and open tools and reuse of open source code modules. This can potentially lower costs and other barriers to entry for first-time developers of mobile applications.

With Python running on Symbian OS, the short development cycle gives a shortcut from the inspiration of an idea to its implementation. It makes rapid prototyping on the mobile platform easy and efficient by wrapping complex, low-level technical details in simple interfaces.

In recent years, the processing power and memory capacity of smartphones have drastically advanced which have made it possible to run an interpreted language such as Python on such devices.

Modern smartphones offer a rich set of features, including WiFi, camera, sound recording and networking that could easily be combined and used for new types of applications. As this book shows, Python for S60 makes accessing these features extremely convenient, letting you focus on your own application idea instead of on the intricacies of the platform.

The mobile space and the Internet are rapidly converging. Client–server solutions can be developed quickly in Python for S60 in combination with a web services back end, such as Django or Ruby on Rails, or using a custom server, which could also be implemented in Python. Being able to use a single agile language, and even some of the same code, on both the client and the server is a great benefit. Chapter 8 deals with advanced topics in networking, such as peer-to-peer communication and turning your mobile phone into a web server. Chapter 9 is dedicated to combining web services, such as Yahoo! Maps and Flickr, with Python for S60.

1.2 How to Use this Book

With a simple text editor and a Nokia smartphone, you can instantly code and test working applications found in this book. You will learn things by running small yet fully working programs on the actual phone to see what they do and then study and modify them on your computer. By experiencing the hands-on coding style and ready-to-use programs, you can soon feel success that keeps you inspired throughout the book.

This book includes over 100 example programs that demonstrate different aspects of the mobile platform. The code for all example programs can be downloaded for free from ***www.mobilepythonbook.com***. Some of the programs are small scripts that show you how to automate tasks, such as sending SMS messages; some are full-scale applications with graphical user interfaces. The examples are designed in such a way that they often build on each other, which makes learning easier through repetition.

Most of the examples demonstrate a specific functionality, so they are short and easy to understand – the median length of examples is just 17 lines of code! However, many of the examples are not just for playing around but are already usable solutions in their own right.

The examples in this book are designed to be combined, modified and enhanced by your own ideas. Throughout the book, examples are cross-referenced with each other, so the book can also be 'dipped into' and not necessarily read from cover to cover.

Besides many examples, Chapters 3 to 6 contain some Python language lessons. These lessons, spanning at most a page, introduce you to the basic concepts of the Python programming language. They provide you necessary knowledge on the language, so you can follow examples and extend them by yourself. Even though the lessons cover the basic concepts in Python, they omit many of its interesting and useful features. As the lessons deal with Python in general and not specifically about Python on the mobile platform, you can easily find more information about the topics in many books and Python tutorials on the web (see the References section).

However, you will be surprised how far you get with 14 one-page lessons on Python!

1.3 Who Is this Book For?

Since Python is easy to learn, you do not need to master any advanced computing concepts before touching this book. You only need an understanding of some basic programming principles or a scripting language, such as PHP or JavaScript, to get started with programming in Python for S60.

Because of the steep learning curve of most mobile platforms, the creative community and novice programmers have been excluded from developing their own ideas for applications for mobile phones. We believe that Python for S60 remedies this problem. Therefore, this book is primarily aimed at people who are new to mobile programming, who lack the time and enthusiasm to learn C++ or Java or who cannot afford to spend weeks or months on development. Rapid prototyping with Python for S60 gives them a fast entry ticket. At the same time, many experienced developers find Python a refreshingly agile alternative and may enjoy the additional sense of elegance and freedom that it provides.

We think the following groups of people will benefit from this book:

- Lead users and 'prototypers'
  If you want to gain knowledge and practical skill for quickly programming working prototypes of innovative mobile applications, you may find Python for S60 your toolkit of choice. It is open source, so you will not be hindered by closed, proprietary platforms which severely restrict your freedom to experiment. If you are an enthusiastic mobile phone user who has many ideas on new ways of using your phone, we show you how to realize your own novel concepts in practice.

- Mobile artists and mobile interaction designers
  Python for S60 will open the door for you to the world of programmable mobile phones. As a creative media artist or interaction designer you might be less constrained by conventional thinking than a typical software engineer – and this is your opportunity. By using Python for S60 to combine several smartphone features, for instance, camera, sound recording, SMS and Bluetooth, you can explore new frontiers of art and design. If you are a designer who has worked with ActionScript for Flash or Director's Lingo and want to start creating mobile applications, you will find Python for S60 already familiar to you in many respects.

- Web developers
  If you have worked with PHP or JavaScript but haven't used Python before, start today! Python for S60 allows you to quickly write mobile client applications that can be part of your website or service. You can also create novel mashups that combine information from a web service with that from your physical surroundings.

- Experienced mobile application developers
  Converting to Python for S60 makes development feel light, happy and productive while retaining your old powers. Whenever a colleague complains that Python is slow or it misses feature X, sit down and write a C++ extension for Python in a few hours. This way, you get the best of both worlds.

- Researchers
  Python for S60 is a perfect platform for doing various kinds of research. It is the easiest way to collect rich empirical data with mobile phones and you can prototype novel applications quickly. Since it is open source and easily extendable in C++, you can even perform some demanding computations on the device. Moreover, you can get started right away, since Nokia smartphones are ubiquitous, off-the-shelf products and Python for S60 is freely available.

- Teachers and students
  If you are teaching introductory programming classes in Python, this book might serve as a source of motivation and inspiration for your students. The smartphone is a rich and ready-to-use platform with many built-in functionalities such as networking, camera, graphics, image handling, GUI design, Bluetooth, telephony and more, so a plethora of concepts can be demonstrated and experimented on it. Nowadays, many students are motivated by the mobile phone, which is an integral, personal part of their lives, rather than by a PC. Being able to easily 'pimp up' the mobile phone and learn programming at the same time is a strong incentive even for younger students.

- Python community
  If you are one of the hundreds of thousands of Python programmers and want to enter the mobile space using completely free and open tools as well as open-source code modules, Python for S60 is an ideal path for you to take.

1.4 What Are Symbian OS, S60 and Python for S60?

Symbian OS is an operating system designed for mobile devices. It includes associated libraries, user interface frameworks and reference implementations of common tools. As a descendant of Psion's EPOC, it runs exclusively on ARM processors. Symbian OS APIs are publicly available and anyone can develop software for Symbian OS.

S60 is a software platform for mobile phones based on Symbian OS. It is Nokia's user interface framework that runs on all Nokia S60 devices on top of Symbian OS. S60 is one of the leading smartphone platforms in the world. It is developed by Nokia, which licenses it to other

manufacturers including Lenovo, LG Electronics, Panasonic and Samsung. S60 consists of a suite of libraries and standard applications based on Symbian OS APIs.

Python is a dynamic object-oriented, open-source, computer-programming language. It can be used for many kinds of software development, for instance, to create stand-alone programs, scalable server software or small scripts – Python's roles are virtually unlimited. Python was created by Guido van Rossum and is distributed under an OSI-approved, open-source license that makes it free to use, even for commercial products.

Python is often used for prototyping and teaching introductory programming classes. It can be learned in a few days and offers strong support for integration with other languages and tools. Python comes with an extensive standard library, thus its slogan is 'Python – batteries included'.

Python runs on most common and legacy platforms, for example, Windows, Mac OS X, Linux/Unix, OS/2, Amiga and Palm OS. It also runs on Nokia S60 2nd and 3rd Edition mobile phones – that is where this book comes in. Python has also been ported to the Java and .NET virtual machines. It is an interpreted programming language that combines remarkable power with clear syntax; it has modules, classes, exceptions, high-level dynamic data types and dynamic typing.

Python for S60 brings the Python programming language to the S60 platform. Python for S60 is based on Python version 2.2.2. It supports many of the Python Standard Library modules but also includes several modules specific to the mobile platform, for example, native GUI elements, Bluetooth, networking, GSM location information, SMS messaging, access to the camera, and more. The full range is described in detail throughout this book. Nokia makes Python bindings for Symbian OS APIs that are publicly provided on S60 devices. All examples in this book were made using Python for S60 version 1.4.0.

1.5 Python Terminology in this Book

In this book, the term 'Python' may refer to three different concepts (see Figure 1.2).

Figure 1.2(a) shows the Python programming language, which is the same both on a PC and a phone, although the PC cannot access the phone's functionalities.

Figure 1.2(b) shows Python for S60, which runs, or interprets, the Python language on the S60 smartphone platform and provides interfaces to the phone's functionalities.

Figure 1.2(c) shows a Python interpreter that is used to run Python on a PC. In some examples of this book, you need that as well.



(a) (b) (c)

Figure 1.2 Three meanings of 'Python': (a) Python code, (b) Python for S60 and (c) Python interpreter on a PC

In practice, almost always in this book we are talking about Python for S60 and thus we use the abbreviation 'PyS60' for clarity. The Python language lessons talk about Python the programming language, so they apply on a PC as well. On rare occasions we need to run the Python interpreter on a PC – in these cases we state this clearly in the text.

1.6 Democratizing Innovation on the Mobile Platform

Eric von Hippel, Professor and Head of the Innovation and Entrepreneurship Group at the MIT Sloan School of Management, discusses in his book ([von Hippel 2005]) the phenomenon that users can generate innovation if a toolkit – based on a platform product – is provided, that allows them to create user-developed modifications that suit their own needs. He calls this 'distributed innovation' by 'lead users'. Lead users have the following characteristics:

- They are ahead of most users in their population with respect to an important market trend and so are experiencing needs today that will later be experienced by many other users.
- They know and understand their own needs well.
- They are close to 'real situations', so the products they develop will appeal to others too.
- They may innovate if they want something that is not available on the market.

Although his empirical data is collected from other fields, his arguments around his lead-user theory matches well with what we practically experience from outcomes of workshops with creative students learning

Python for S60. We see that they innovate instantly, creating unusual and novel applications based on their own ideas, fulfilling their own needs and enabling them to share their innovations.

Can Python for S60 democratize innovation on the mobile platform? We have the vision of a big garden full of beautiful flowers, each representing a novel mobile application created by a lead user fitting his or her own needs. Will you help to grow these flowers? Distributed innovation on the mobile platform is possible and perhaps this book can be a starting point to help trigger it.

Let's look now at some of the topics and arguments that [von Hippel 2005] states and how they map to Python for S60.

1.6.1 User-Centered Innovation Process

[von Hippel 2005] explains that users that innovate can develop exactly what they want, rather than relying on manufacturers to act as their (often imperfect) agents. It may be that the needs of local user communities differ and so local lead users really may be the world's lead users with respect to their particular needs. Further, [von Hippel 2005] argues that users generally have a more accurate and more detailed model of their needs than manufacturers. The information assets of some particular users are close to what is required to develop a particular innovation. Users tend to develop innovations that are functionally novel, requiring a great deal of user-need information and use-context information for their development.

Using Python for S60, users can program applications based on their interests and own ideas – even integrating local cultural aspects. Users with few programming skills can innovate, iterating through new ideas rapidly.

1.6.2 Motivation of Lead Users

[von Hippel 2005] also states that, for individual user–innovators, enjoyment and learning of the innovation process can be important.

To program with Python for S60 is often described by people as fun since it generates reward and motivation through a seamless process of iterative development and design with instant coding, modifying and testing on the real phone in the real mobile network. People can easily learn how to start coding their own mobile application ideas with hardly any learning curve.

1.6.3 Sharing of Innovations

According to [von Hippel 2005], users often achieve widespread diffusion: they often 'freely reveal' what they have developed. Individual users

can benefit from innovations developed and shared by others. Freely revealing users may benefit from enhancement of their reputation from positive network effects because of increased diffusion of their innovation.

Coding Python for S60 modules and making them public for sharing with others is becoming a common practice. The projection is for huge potential if many lead users come on board to contribute.

1.6.4 Development of Products by Lead Users

[von Hippel 2005] states that studies have shown that many of the innovations reported by lead users are judged to be commercially attractive or have actually been commercialized.

Python for S60 allows lead users and creative minds to prove the concept of their own ideas, ideas that fulfill some real needs and can potentially be shared with others. In our experience from giving mobile phone programming workshops around the world, when we ask people what they would like to do with their phone, almost everyone comes up with a unusual idea.

1.6.5 Toolkit

Further, [von Hippel 2005] says that the ability of users 'to innovate is improving radically and rapidly through improved access to easy-to-use tools and steadily richer innovation commons. Companies learn to supply proprietary platform products that offer user–innovators a framework upon which to develop and use their improvements. . . . kits and design tools . . . can serve as platforms upon which to develop and operate user-developed modifications.'

With Python for S60, a toolkit provides the S60 platform with many open APIs and a rich set of features and phone functionality into which creative users can tap. With a hands-on tutorial, such as MobiLenin, lots of starting code and the help of this book, people can program their own ideas quickly and in a powerful manner.

1.7 The Process of Rapid Prototyping with Python S60

Python for S60 can be seen as an ideal prototyping tool. Turning an idea or concept into code for a working software prototype can be done in weeks, if not days. Many fully functioning code examples, such as the ones found in this book, can be used as a springboard to get started with the rapid prototyping process. Rapid prototyping with Python for S60 may:

- save much development time
- save many development costs
- allow a developer team to turn several ideas into prototypes within budget and time limits, instead of building just one.

1.8 Summary

Python for S60 is all about having one's head in the clouds, one's hands in mud and one's feet on the ground. In other words, it allows you to come up with ideas and implement and test them in a straightforward and pragmatic manner. Dive in, develop, share and enjoy!

In the next chapter we take the first practical steps. We start by installing the Python for S60 interpreter on your phone and then create our first PyS60 program.

2

Getting Started

In this chapter we show you, step by step, how to install Python for S60 (PyS60) to your mobile phone. Once this is done, you can make your first PyS60 script and run it on your phone.

Since PyS60 (Figure 2.1) is not pre-installed on any S60 mobile phone by default (at least not when this book was written), you have to install it on your device. PyS60 is available on the Internet for free download.

To use PyS60, you need:

- a Nokia mobile phone, based on S60 2nd Edition or 3rd Edition
- a memory card for the phone



Figure 2.1 Python for S60

- a computer which runs Windows, Mac OS X, or Linux
- a USB cable or Bluetooth to connect the phone to your computer.

If you are not sure whether your mobile phone uses 2nd Edition, 3rd Edition or some earlier version of the S60 platform, you can find out in Appendix D. Table D.1 includes information about all Nokia phone models that are based on the S60 platform at the time of writing this book. All models released after this book will use either 3rd Edition or some newer edition of the S60 platform. If you have a phone model that predates this book (2007) and it is not included in the table, it is unlikely to be compatible with PyS60.

If your phone uses S60 3rd Edition, follow the instructions in Section 2.1. If your phone uses S60 2nd Edition, follow the instructions in Section 2.2.

The descriptions given here are subject to change as PyS60 develops. If they do not work, you may check out the latest instructions on the Internet from relevant websites such as ***http://wiki.forum.nokia.com***, ***www.mobilepythonbook.com*** and ***www.mobilenin.com***. Python is a fast evolving language.

2.1 Installing Python for S60 on 3rd Edition Devices

2.1.1 Download the Installation Files

The first task is common to all platforms. You need to download two installation files from the Internet to your computer and then to the phone. You find them at SourceForge's PyS60 project page, ***http://sourceforge.net/projects/pys60***. There are many versions of files available on that website. The two files that you need are the PyS60 interpreter and the user interface (Python Script Shell) for the PyS60 interpreter.

The file names consist of three components:

- the name
- the version number, for example, 1.4.0 (make sure you choose the latest version, that is, the largest number)
- the edition number (in this case, 3rdEd).

Assuming that version 1.4.0 is the latest version, you will download `PythonForS60_1_4_0_3rdEd.SIS` (the PyS60 interpreter) and `PythonScriptShell_1_4_0_3rdEd.SIS` (the user interface). Both files have undergone the signing process that is required for 3rd Edition devices (see Appendix A for more information regarding signing).

We have divided the following descriptions into separate sections for

- Windows users, see Section 2.1.2
- Mac users, see Section 2.1.3
- Linux users, see Section 2.1.4.

For each operating system, we have included a complete description of how to install PyS60 and how to write and execute your first script.

2.1.2 Installation for Windows Users

We assume you have the Nokia PC Suite application installed on your Windows computer. It connects your Nokia phone to your Windows PC for fast file transfer and smooth synchronization. If you have not installed it yet, we recommend doing so now. Most new phones come with the Nokia PC Suite software on a CD. You should download the latest version from Nokia's website, for example, at ***www.nokia.com/A4144903***. If you do not have Nokia PC Suite available or you have no USB cable, check Sections 2.2.and 2.5.1 for alternatives.

1. Install the downloaded .SIS files to your phone

Connect your phone to your computer using the USB cable. Open the Nokia PC Suite and select 'Install applications' (see Figure 2.2). First, install the PyS60 interpreter and then the user interface for the PyS60 interpreter. Follow the on-screen instructions on the phone – for instance, select 'Yes' if a security warning appears on the phone screen. Also make sure the date is set correctly on the phone.

After the installation is complete, your phone shows the Python icon on the desktop or inside one of the desktop folders (Figure 2.1). Your phone is now ready to execute Python scripts. Let's prepare a 'Hello world' script.

2. Write a Python Script

You can write a Python script on your computer with any text editor. Useful editors are, for example, ConTEXT or PythonWin which are freely downloadable on the Internet, but Notepad also works. Symbian developers can use Carbide.c++ with the PyDev plug-ins found at ***http://pydev.sourceforge.net***.

Write the following line in your text editor:

```
print "Hello world!"
```



Figure 2.2 Nokia PC suite's main window

After the code is typed, save the file under the name `hello.py`. Make sure that the file ending is .py and not .txt.

The file is now ready to be executed on your phone! You do not have to build or compile it any way. However, it must be copied to the phone first.

3. *Upload a Python Script to a Phone*

Upload your Python script to the phone with Nokia PC Suite's file manager (Figure 2.2). Create a folder named Python on the phone's memory card (drive E: in Figure 2.3). Then copy the `hello.py` file from your computer to the `E:\Python` folder on the phone. Now the script is ready to be tested!

4. *Test a Python Script*

Start the PyS60 interpreter by clicking on the Python icon (Figure 2.1) on the desktop or inside the appropriate folder on your phone. Once the PyS60 interpreter has started up, select 'Options' (Figure 2.4(a)) and 'Run script' (Figure 2.4(b)). Choose your script name, `e:hello.py`, from the list and select 'OK'. Your script should now start up and you should see a green line stating Hello world! (Figure 2.5).

Congratulations! You have successfully written and executed your script with PyS60! To go through all the examples in the book, repeat steps 2–4 for each new script. Have fun with it!

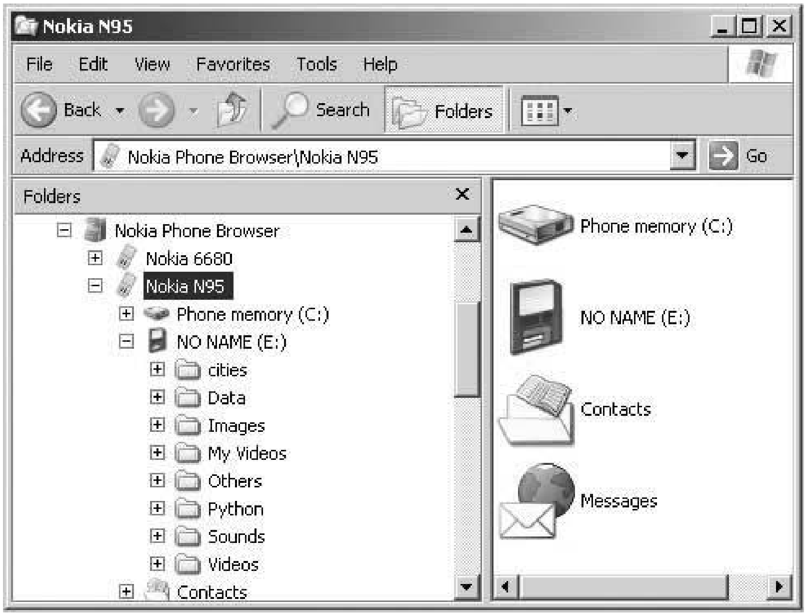
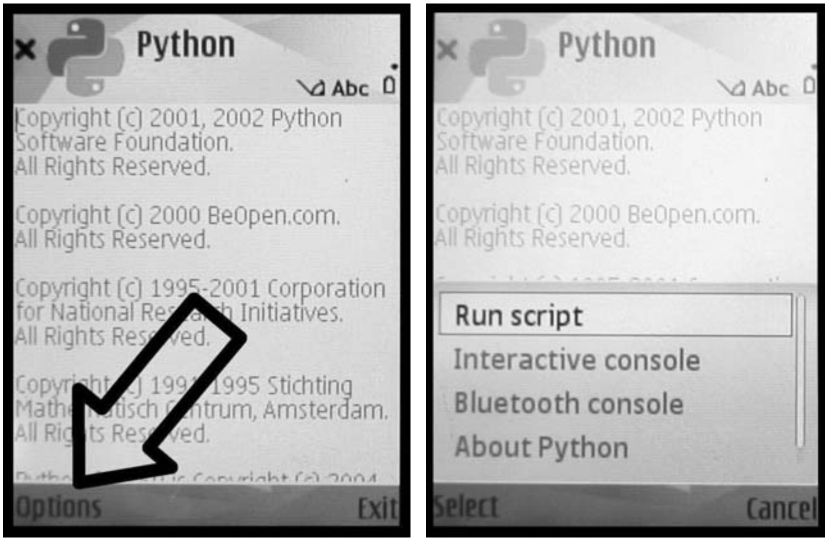


Figure 2.3 File manager



(a) (b)

Figure 2.4 Python running on a phone

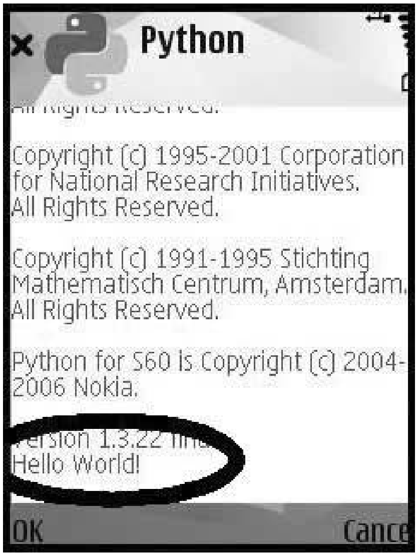


Figure 2.5 Our script running on the phone

2.1.3 Installation for Mac OS X Users

1. Install the downloaded .SIS files to your phone

Connect your phone to your computer using the USB cable, then select 'Data transfer' or 'Mass storage' mode on the phone screen. The memory card of the phone is mounted as an external 'hard drive' and its content can be accessed by the Finder application on the Mac. You must have the memory card inserted into your phone.

Copy the downloaded installation files to any folder on the memory card of your phone, for example, to the root. Safely remove (un-mount) the phone's drive from your computer in the same way as with any external hard drive.

On the phone, open the File Manager application (usually found on the desktop of the phone or inside a subfolder e.g. 'Tools') and go to the directory of the memory card where you have stored the installation files. Click on the installation files. First, install the PyS60 interpreter (PythonForS60) and then the user interface for the PyS60 interpreter (PythonScriptShell). Follow the on-screen instructions on the phone – for instance, select 'Yes' if a security warning appears on the phone screen. Also, make sure the date is set correctly on the phone.

After the installation is complete, your phone shows the Python icon on the desktop or inside one of the desktop folders (Figure 2.1). Your phone is now ready to execute Python scripts. Let's prepare a 'Hello world' script.

2. *Write a Python Script*

You can write a Python script on your computer with any text editor. Useful editors are, for example, SubEthaEdit, TextMate or BBEdit. We do not recommend using Mac's TextEdit application, as it might place some invisible characters in your code which throw an error when executing the script.

Write the following line in your text editor:

```
print "Hello world!"
```

After the code is typed, save the file under the name `hello.py`. Make sure that the file ending is .py and not .txt.

The file is now ready to be executed on your phone! You do not have to build or compile it any way. However, it must be copied to the phone first.

3. *Upload a Python Script to a Phone*

Again, connect the phone to your computer as an external 'hard drive'. Create a folder called Python on the phone's memory card (drive E:) using the Mac's Finder application. Copy the `hello.py` file from your computer to the `E:\Python` folder on the phone.

Safely remove (un-mount) the phone as an external hard drive from the computer. You can now test the script.

If your Mac has built-in Bluetooth, you can use it to upload the `hello.py` file to your phone. Click on the Bluetooth icon at the top of the Mac screen (see Figure 2.8), select 'Browse device' and choose your phone's nickname from the list (Bluetooth must be switched on on the phone). Then select 'Browse' to open a window showing the phone's memory card `E:\`. Drag and drop `hello.py` from the location in your Finder application to the `E:\Python` folder in the newly opened window. This saves you having to use the USB cable and mount and un-mount the phone as an external 'hard drive'.

4. *Test a Python Script*

Start the PyS60 interpreter by clicking on the Python icon (Figure 2.1) on the desktop or inside the appropriate subfolder on your phone. Once the PyS60 interpreter has started up, select 'Options' (Figure 2.4(a)) and 'Run script' (Figure 2.4(b)). Choose your script name, `e:hello.py`, from the list and select 'OK'. Your script should now start up and you should see a green line stating Hello world! (Figure 2.5).

Congratulations! You have successfully written and executed your script with PyS60! To go through all the examples in the book, just repeat steps 2–4 for each new script. Have fun with it!

2.1.4 Installation for Linux Users

We assume that your computer has either a built-in Bluetooth capability or an external Bluetooth dongle. Also, we assume that Bluetooth is detected correctly by your distribution and you have the `obexftp` and `hcitool` programs installed. In many distributions, they can be installed from packages `obexftp` and `bluez-utils`.

1. Install the downloaded .SIS files to your phone

First, enable Bluetooth on your phone. Then, on the command line, execute the following command:

```
hcitool scan
```

This performs Bluetooth scanning. You should see the name of your phone in the list, with its Bluetooth address (e.g., 00:17:ED:AC:56:FE). Next, you can upload the SIS files to your phone with the following commands:

```
obexftp -b 00:17:ED:AC:56:FE -c E: -p PythonForS60...SIS
obexftp -b 00:17:ED:AC:56:FE -c E: -p PythonScriptShell...SIS
```

Replace the Bluetooth address above with your phone's actual Bluetooth address. Also, replace the file names with the actual names of the files that you downloaded.

On the phone, open the File Manager application (usually found on the desktop of the phone or inside a subfolder e.g. 'Tools') and go to the root of the memory card. You should see the two files there. Click on the installation files. First, install the PyS60 interpreter (PythonForS60) and then the user interface for the PyS60 interpreter (PythonScriptShell). Follow the on-screen instructions on the phone – for instance, select 'Yes' if a security warning appears on the phone screen. Also, make sure the date is set correctly on the phone.

2. Write a Python Script

You can write a Python script on your computer with any text editor. Useful editors are, for example, Vim or Emacs.

Write the following line in your text editor:

```
print "Hello world!"
```

After the code is typed, save the file under the name `hello.py`. Make sure that the file ending is .py and not .txt.

The file is now ready to be executed on your phone! You do not have to build or compile it any way. However, it must be copied to the phone first.

3. Upload a Python Script to a Phone

We use the same `obexftp` tool for this as in Step 1. Execute the following command on the command line, replacing the Bluetooth address with your phone's actual address:

```
obexftp -b 00:17:ED:AC:56:FE -c E: -C Python -p hello.py
```

4. Test a Python Script

Start the PyS60 interpreter by clicking on the Python icon (Figure 2.1) on the desktop or inside the appropriate subfolder on your phone. Once the PyS60 interpreter has started up, select 'Options' (Figure 2.4(a)) and then 'Run script' (Figure 2.4(b)). Choose your script name, `e:hello.py`, from the list and select 'OK'. Your script should now start up and you should see a green line stating Hello world! (Figure 2.5).

Congratulations! You have successfully written and executed your script with PyS60! To go through all the examples in the book, just repeat steps 2–4 for each new script. Have fun with it!

2.2 Installing Python for S60 on 2nd Edition Devices

We assume that your computer has either a built-in Bluetooth capability or an external Bluetooth dongle. You should have Bluetooth working on your computer.

2.2.1 Download the Installation Files

You need to download two installation files from the Internet to your computer. You find them at SourceForge's PyS60 project page, ***http://sourceforge.net/projects/pys60***. There are many versions of files available on that website. All you need are two files: the PyS60 interpreter and the user interface for the PyS60 interpreter.

The file names consist of four components:

- the name
- the version number, for example, 1.4.0 (make sure you choose the latest version, that is, the largest number)
- the edition number (in this case, 2ndEd)
- the choice of Feature Pack; depending on your phone model, you should choose files ending with FP2 or FP3 (Table D.1 shows which is suitable for your phone).

Assuming that version 1.4.0 is the latest version and your phone uses Feature Pack 2, you download `PythonForS60_1_4_0_2ndEdFP2.SIS` (the PyS60 interpreter) and `PythonScriptShell_1_4_0_2ndEdFP2.SIS` (the user interface).

2.2.2 Send the Installation Files to the Phone

You use Bluetooth to send the files to the phone's message inbox. Make sure you have switched on Bluetooth on your phone (settings can be found in the 'Connect' panel).

Windows Users

In the File Explorer application, go to the location of the installation files. Right-click on the file to send. Select 'Send To' and then 'Bluetooth device' (see Figure 2.6). A window asks you to select where you want to send the file. Select 'Browse' and wait a moment. Your computer will search for nearby Bluetooth devices.

A list (see Figure 2.7) will show the nicknames of all Bluetooth devices found. Select the nickname of your phone and select 'OK' and then 'Next'. A popup note should be displayed on your phone asking 'Receive message by way of Bluetooth . . . ?' and you should select 'Yes'. A message

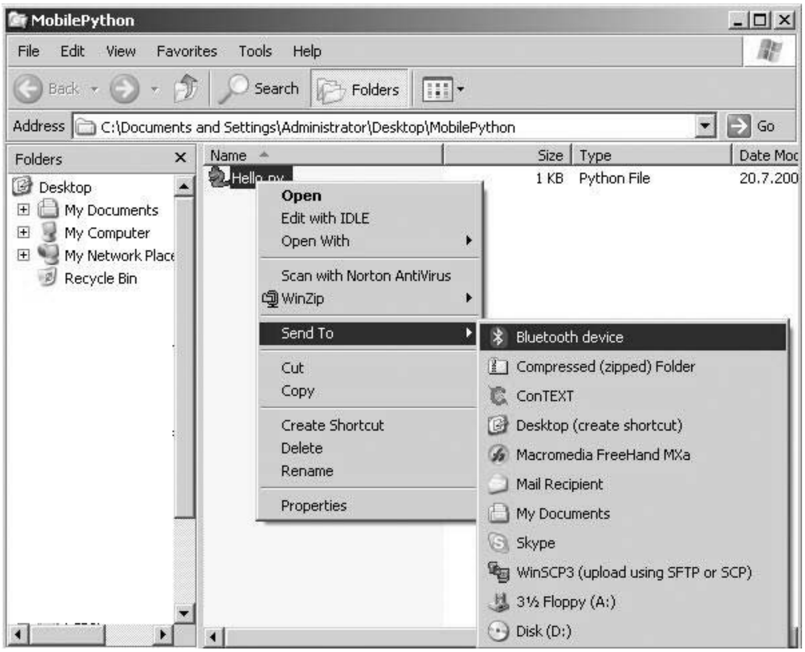


Figure 2.6 File Explorer

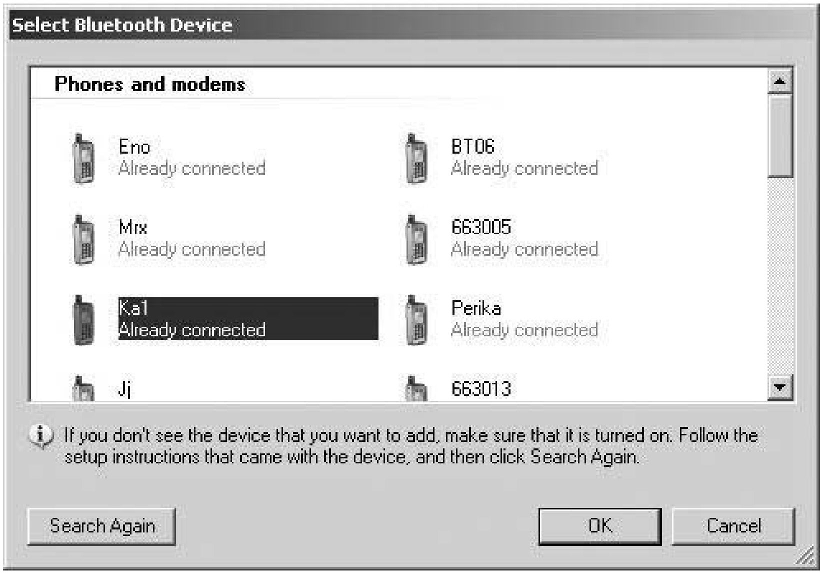


Figure 2.7 Bluetooth devices found

should then arrive in your inbox. Repeat these steps for the second installation file.

In different Windows versions, the menus might look slightly different, but the process is basically the same. Also you might have to do Bluetooth pairing between your phone and your computer before you can send the file. You can find instructions on how to do this in Section 7.1.

Mac OS X Users

Select the Bluetooth icon on the Mac window and select ‘Send file’ (see Figure 2.8). Then select one of the installation files and select ‘Send’. This opens a window where you need to select ‘Search’ to make your computer search for nearby Bluetooth devices.

A list (see Figure 2.9) will show the nicknames of all Bluetooth devices found. Select the nickname of your phone and press ‘Send’. A popup note should be displayed on your phone asking ‘Receive message by way of Bluetooth...?’ and you should select ‘Yes’. A message should then arrive in your inbox. Repeat these steps for the second installation file.

In different Mac OS versions, the menus might look slightly different, but the process is basically the same. Also you might have to do Bluetooth pairing between your phone and your computer before you can send the file. You can find instructions on how to do this in Section 7.1.



Figure 2.8 Initiating Bluetooth transfer on a Mac

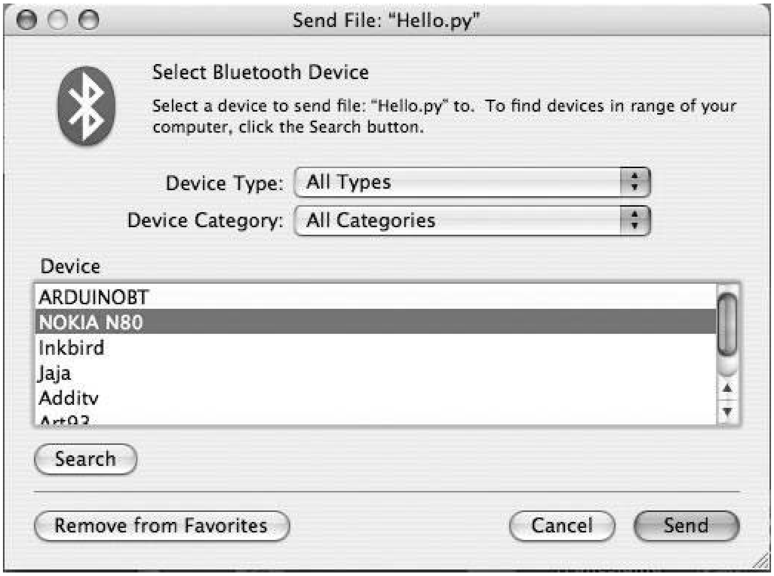


Figure 2.9 Bluetooth devices found

Linux Users

You need the `ussp-push` and `hcitool` programs installed. In many distributions, they can be installed from packages `ussp-push` and `bluez-utils`.

Perform Bluetooth scanning to find the Bluetooth address of your phone:

```
hcitool scan
```

You should see the name of your phone in the list, with its Bluetooth address (e.g., 00:17:ED:AC:56:FE). Next, you can upload the SIS files to your phone with the following commands:

```
ussp-push 00:17:ED:AC:56:FE@ PythonForS60...SIS PyS60.SIS
ussp-push 00:17:ED:AC:56:FE@ PythonShellScript...SIS Shell.SIS
```

Replace the Bluetooth address above with your phone's actual Bluetooth address. Also, replace the file names with the actual names of the files that you downloaded.

After each file has been uploaded, a popup note should be displayed on your phone asking 'Receive message by way of Bluetooth...?' and you should select 'Yes'.

2.2.3 Install the Files to the Phone

Open the received messages by selecting them in order. First, install the PyS60 interpreter (PythonForS60) and then the user interface for the PyS60 interpreter (PythonScriptShell). Follow the on-screen instructions on the phone – for instance, select 'Yes' if a security warning appears on the phone screen. If you are asked to install the files on the phone memory (Ph. Mem) or the memory card (M. card), it is up to you which one to choose.

After the installation is complete, your phone shows the Python icon on the desktop or inside one of the desktop folders (Figure 2.1). Your phone is now ready to execute Python scripts. Let's prepare a 'Hello world' script.

2.2.4 Writing and Running a Python Script

1. Write a Python Script

You can write a Python script on your computer with any text editor. Useful editors for Windows are, for example, ConTEXT or PyWin which are freely downloadable on the Internet, but Notepad also works. Possible editors for Mac are, for example, SubEthaEdit, TextMate or BBEdit. We do not recommend using Mac's TextEdit application, as it might place some invisible characters in your code which throw an error when executing the script. In Linux, you can use Vim or Emacs.

Write the following line in your text editor:

```
print 'Hello world!'
```

After the code is typed, save the file under the name `hello.py`. Make sure that the file ending is .py and not .txt.

The file is now ready to be executed on your phone! You do not have to build or compile it any way. However, it must be copied to the phone first.

2. *Install a Python Script on a Phone*

You send a Python script to the phone in the same way that you sent the installation files in Section 2.2.3.

Open the received `hello.py` file in the message inbox by selecting it. The message inbox handler will recognize it as a PyS60 file and display a popup note. When asked to install your file as a 'Python script' or a 'Python lib module' (Figure 2.10), choose 'Python script'. Your Python script is automatically installed to the correct place. Now you can test the script.



Figure 2.10 Installing a Python script

3. *Test a Python Script*

Start the PyS60 interpreter by clicking on the Python icon (Figure 2.1) on the desktop or inside the appropriate subfolder on your phone. Once the PyS60 interpreter has started up, select 'Options' (Figure 2.4(a)) and

'Run script' (Figure 2.4(b)). Choose your script name, `hello.py`, from the list and select 'OK'. Your script should now start up and you should see a green line stating Hello world! (Figure 2.5).

Congratulations! You have successfully written and executed your script with PyS60! To go through all the examples in the book, just repeat steps 1–3 for each new script. Have fun with it!

2.3 Writing a Program in Python for S60

Example 1: First PyS60 program

We write a simple program consisting of three lines of code as our first real example. It should do the following:

1. Display a text input field on the screen; the instruction in the text field should say 'Type your name' (see Figure 2.11(a)).
2. Display a popup note stating 'Greetings from:' followed by whatever the user typed into the text input field (see Figure 2.11(b)).



(a) (b)

Figure 2.11 Our first Python script example: (a) a text input field and (b) a popup note

The code to do this is as follows:

```
import appuifw
name = appuifw.query(u"Type your name:", "text")
appuifw.note(u"Hello world! Greetings from: " + str(name), "info")
```

In the first line of code, we import the appuifw module, which handles user interface elements such as text input fields and popup notes.

In the second line of code, we create a single-field dialog (Figure 2.11(a)) using the query() function of the appuifw module with two parameters: *label* and *type*. For the first parameter, *label*, we put the text u"Type your name:". The u is required because the phone understands only text declared as Unicode and the quotation marks are needed because the *label* parameter must be given as a string. As the second parameter, *type*, we put "text", to declare the input field as a text dialog. Other possible types are "number", "date", "time", "query" and "code". The two parameters are separated by a comma. This line of code also creates a variable called *name* which will receive the result of the text input field after the user has typed something into the dialog.

In the third line of code, we create a popup note (Figure 2.11(b)) using the note() function of the appuifw module with two parameters: *label* and *type*. For the first parameter, *label*, we put the text u"Hello world! Greetings from: "+ str(name). This is the text that will appear inside the popup note. Again, the string must be given in quotation marks but the popup note will also include the result that the user has typed in, so we add the contents of the variable *name* to the string. As the second parameter, *type*, we put "info", to declare the popup note as information; that causes a green exclamation mark to appear inside the popup note. Other possible types are "error", showing a red exclamation mark, and "conf"(confirmation), showing an animated check mark. The two parameters are separated with a comma.

Type this example into a file and upload it to your phone for testing, by following the instructions in Sections 2.1 or 2.2.

2.4 White Space in Python Code

Python does not use curly brackets or special begin and end statements to structure code. Instead, the indentation level of a line determines to which *code block* the line belongs. A code block is a group of statements that belongs to an if clause, a for-loop, a function or a similar structure. Indentation starts a code block and it ends when the indentation stops.

In the following example, the first two print statements form a code block that belongs to the if clause. The last print statement does not belong to the code block and it is executed regardless of the if condition.

```
if x > 5:
    print "X is greater than 5"
    print "Also this line belongs to the if-clause"
print "This line does not belong to the if-clause"
```

You can use spaces or tabs for indentation but you must be consistent. In this book, we use four spaces for indentation. You can follow the same style in your own code. Many text editors that support Python programming, such as Emacs or PythonWin, automatically make sure that the indentation is consistent. Using an editor like this is highly recommended.

In some cases you must split a long expression over several lines. As white space is significant only at the beginning of a statement, you can split a long expression freely over several lines. However, in some cases the Python interpreter cannot know whether the subsequent lines belong to one statement or several statements. To make your intention clear, you can put a backslash character '\' at the end of a line to specify that the next line continues the same statement.

2.5 Troubleshooting

To avoid an installation error, make sure the date is set correctly on the phone, since 'signed' installation files have a date that specifies from when they work. With some phone models, you might encounter an error such as 'Certificate error. Contact the application supplier'. You can fix this by selecting 'Tools, App. mgr, Options, Settings'. Change 'Software installation' from 'Signed Only' to 'All' and 'Online certif. check' to 'Off'.

2.5.1 Bluetooth as an Alternative to a USB Cable

Instead of using a USB cable to connect your phone to a computer, you can use Bluetooth, if your PC hardware has built-in Bluetooth functionality or you have a USB Bluetooth dongle available. Check the manual of the Nokia PC Suite for instructions, as well as your phone's manual for how to set up the connection.

2.5.2 Bluetooth Fails

You might need to pair your phone with your computer. See the phone manual to find out how to do this. Typically, pairing can be performed by way of the phone's 'Connectivity, Bluetooth' settings.

2.5.3 No Nokia PC Suite Available

If you do not have Nokia PC Suite installed, you can connect your phone to the computer as an external hard drive. Please check Section 2.1.3 for instruction on how to get this to work (it works in a very similar way on both Windows and Mac).

2.5.4 Using an Emulator

Testing scripts with an emulator on a PC is possible. However there are some restrictions when using an emulator, compared to using an S60 phone. For example, you cannot send and receive SMS messages or take a photo with an emulator. Check Appendix D for instructions on how to set up the emulator.

2.6 Summary

Jumping head first into hands-on action, we showed you in this chapter how to write your first script with PyS60 and how to run it on your S60 mobile phone. We covered the basic steps to do this:

1. Download the installation files from the Internet.
2. Install the downloaded .SIS files to your phone.
3. Write a Python script on your computer.
4. Upload your Python script to the phone.
5. Test your script on the phone.

In Section 2.3, we programmed our first PyS60 application with three lines of code. We hope this example got you excited to discover all the great things you can do with PyS60. By continuing with a practical hands-on approach, we guide you through the next chapter, helping you to unfold your creativity and enabling you to program applications based on your own ideas.

If you feel really bold and adventurous now, you can see an alternative approach for updating PyS60 scripts on the phone over a network connection in Section 10.3.3. For some developers, this might be the most productive way to use PyS60. If you want to see even more nifty tricks that PyS60 can do, Appendix B tells you how to use the PyS60 interpreter remotely on your PC over Bluetooth.

3

Graphical User Interface Basics

The native graphical user interface elements are some of the easiest to learn of the features that Python for S60 offers. We cover them here at the beginning of the book, to give you a smooth start in learning PyS60.

We explain the graphical user interface (UI) elements using small exercises. Each exercise includes instructions, screenshots and detailed code descriptions. These exercises should help you in understanding and running the examples presented in the subsequent chapters. We also cover some Python language lessons, which will help you to get to know the Python programming language by heart, through learning by doing.

In Example 1 (see Chapter 2), we introduced two native graphical user interface elements: the text input field (a single-field dialog) and the popup note. Here we want to show a whole range of them:

- note – popup notes
- query – single-field input dialog
- multi-query – two-field text input dialog
- popup menu – simple menu
- selection list – simple list with find pane
- multi-selection list – list to make multiple selections

In the following paragraphs, we look at these elements in detail and explain how you can use them in your own programs.

3.1 Using Modules

The native UI elements that PyS60 offers are accessible through a module called `appuifw` (which stands for application user interface framework).

It is an interface to the S60 UI application framework that takes care of all UI functionalities on the S60 platform. But before we go on, let's learn what a 'module' is in Python, in the language lesson.

Python Language Lesson: module

A module is a file that contains a collection of related functions and data grouped together. PyS60 comes with a rich set of modules, for example `messaging` to handle SMS functionalities, `camera` for taking photos, and `appuifw`, which provides ready-to-use UI elements. The modules' contents are described in detail in the Python Library Reference and Python for S60 API documentation. We also learn about many of them in this book.

To use a module in your code, it must be imported at the beginning of the script, for example:

```
import appuifw
```

To address a function contained in a module, we first write the module name, a dot and then the function name:

```
appuifw.query(label, type)
```

Here, `appuifw` is the module name and `query` is the function we want to use.

You may import many modules using a single import statement:

```
import appuifw, e32
```

This imports two modules: `appuifw` and `e32`.

As you might remember from our Example 1 (Chapter 2), we used the functions `query()` and `note()`, which belong to the `appuifw` module. These functions generate UI elements, dialogs, that are displayed on the screen when the PyS60 interpreter executes the script. They become visible on the phone screen as soon as the corresponding Python function is called.

3.2 Native UI Elements – Dialogs, Menus and Selection Lists

Let's now look in detail at some of the native UI elements or dialogs that PyS60 offers.

3.2.1 Single-Field Dialog: query

Syntax: `query(label, type[, initial value ])`
Example code:

```
appuifw.query(u"Type a word:", "text", u"Foo")
```

This function shows a single-field dialog. The dialog can include some instruction text that is passed as a string (by putting the u in front of the string) to the parameter, `label`. The type of the dialog is defined by the parameter, `type`. The value of `type` can be any of the following strings: `"text"`, `"number"`, `"date"`, `"time"`, `"code"`, `"query"` or `"float"` (see Figure 3.1).

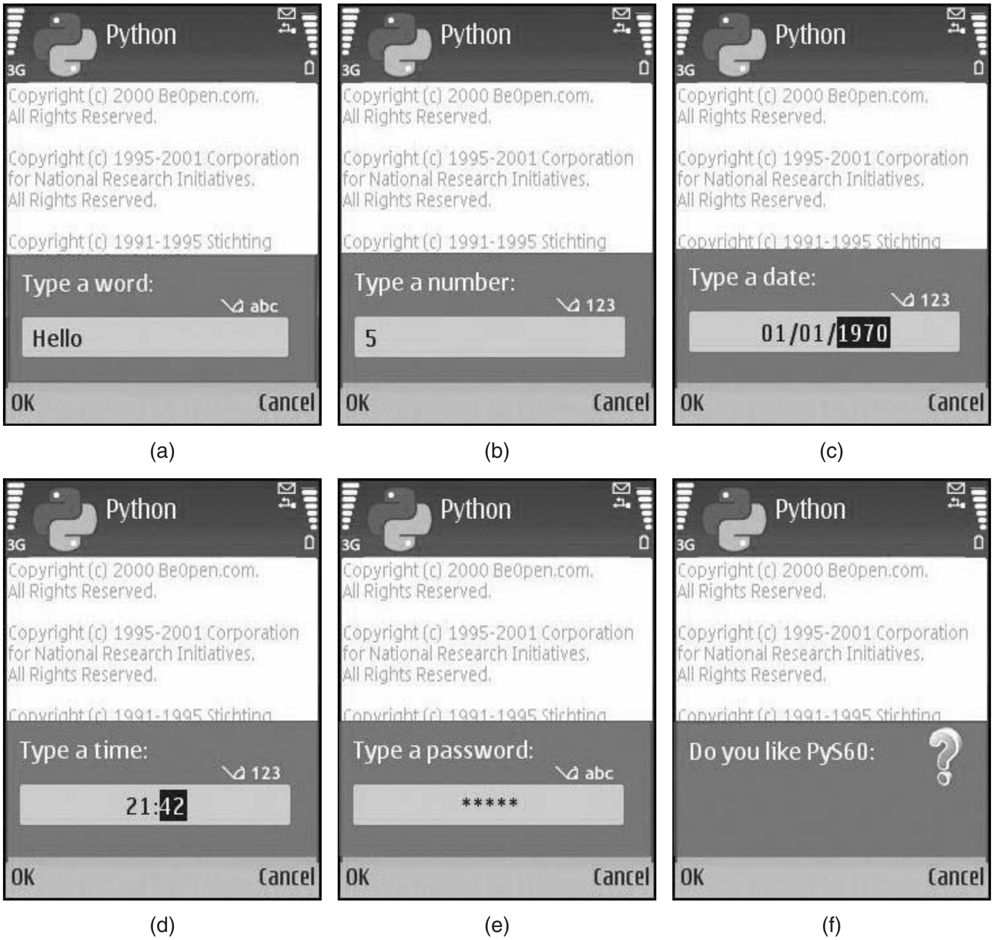


Figure 3.1 Types of single-field dialog: (a) text, (b) number, (c) date, (d) time, (e) code and (f) query

An initial value can be included, so the dialog shows a default input value when it appears on the phone screen. This is done simply by adding the initial value as an input parameter to the function, for example:

```
appuifw.query(u"Type a word:", "text", u"Foo")
```

The return value of the `appuifw.query()` function depends on the `type` parameter:

- For text fields (types of 'text' and 'code'), the return value is a Unicode string.
- For number fields, it is an integer.
- For date fields, it is the number of seconds since the epoch (0:00 on 1 January 1970, UTC) rounded down to the nearest local midnight.

Exercise

To see how the various single-input dialogs appear on your phone, type the lines of code in Example 2 into your text editor and save the file with a name ending .py. Transfer the .py file to your phone or to the emulator as described in Chapter 2. Start the PyS60 interpreter on the phone and select 'Options'. Select 'Run script', choose your script from the list, select 'OK' and see what happens.

You should see all the single-input dialogs as in Figure 3.1 appear on your phone's screen one by one. Each dialog will wait for you to type something in and select 'OK'.

Example 2: Various dialogs

If you want, change the instruction text (the `label` parameter) of the `query` function. As you can see, we need to import the `appuifw` module at the beginning of the script to make this work.

```
import appuifw
appuifw.query(u"Type a word:", "text")
appuifw.query(u"Type a number:", "number")
appuifw.query(u"Type a date:", "date")
appuifw.query(u"Type a time:", "time")
appuifw.query(u"Type a password:", "code")
appuifw.query(u"Do you like PyS60", "query")
```

Since Symbian OS and the S60 platform are used all over the world, they need to be able to show text written in various languages and writing systems. Unicode provides a consistent way to encode text written in any writing system. In order to be globally compatible, all text-related

functionalities in S60 and Symbian OS accept only Unicode strings. In Python, the 'u' character in front of a string denotes that the text is written in Unicode. If you see strange boxes on the screen instead of characters, you have probably forgotten to put 'u' in front of the corresponding text. We talk more about Unicode conversions in Section 6.2.

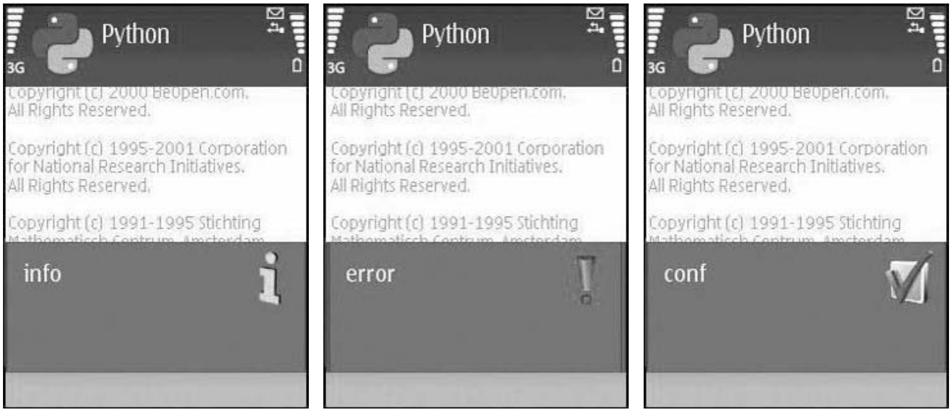
3.2.2 Note Dialog: note

Syntax: `note(text[, type[, global ] ])`
Example code:

```
appuifw.note(u"upload done", "conf")
```

This function displays a note dialog (a popup note) on your phone with user-specified text. In the example, it displays the words 'upload done'. This is achieved by passing the Unicode string 'upload done' as the input parameter `text`.

There are three possibilities for `type`: `"info"`, `"error"` and `"conf"` (see Figure 3.2). Each type shows a different symbol inside the popup note dialog: 'info' shows a green exclamation mark after the text and 'error' shows a red exclamation mark. The default value for `type` is `"info"`, which is automatically used if `type` is not set.



(a) (b) (c)

Figure 3.2 Types of note dialog: (a) information, (b) error, (c) confirmation

Exercise

To see how the popup notes appear on your phone, type in the lines of code shown in Example 3 and run your script on your phone or the emulator to see what happens.

You should see all the different popup notes appear on your phone screen one by one.

Example 3: Various notes

Note again that we need to import the `appuifw` module at the beginning of the script in order make this work.

```
import appuifw
appuifw.note(u"Hello")
appuifw.note(u"File not found", "error")
appuifw.note(u"Upload done", "conf")
```

Python Language Lesson: variable

A variable is a name that refers to a value. In Python you do not have to declare variables, that is, you do not have to specify in advance the type, such as integer, string, and so on. Variables are implicitly typed, meaning that the type of a variable is the type of the value it refers to.

If your variable holds the integer 5 but you need the string '5', you have to use the `str()` function to convert the integer to a string. The type of a value does not change automatically.

The first time you use a variable, assignment is done with the equals sign '='. It creates a new variable and gives the variable a value, for example:

```
data = 5
mynewlist = "Symbian"
```

The special value `None` denotes a missing value. You can re-use the same variable freely throughout a program.

In some examples, we use capital letters for variable names. This is a common way in Python to denote variables whose values must not change (that is, they are constants), as a convenience to developers. The interpreter itself does not care if they are constant.

3.2.3 Multi-Query Dialog: multi_query

Syntax: `multi_query(label1, label2)`
Example code:

```
appuifw.multi_query(u"First name:", u"Last name:")
```

This function displays a two-field text input dialog. The input parameters `label1` and `label2` can be pre-defined by inserting Unicode strings.

The `multi_query()` function returns the values that the user inputs. The values of the upper and lower input fields are jointly returned as a *tuple*. In Python, a tuple is a comma-separated list of values, for example (data1, data2). We learn more about handling tuples in Chapter 4. This function, as with other dialogs, returns the empty value `None` if the user cancels the dialog.

Exercise

To see how the multi-query dialog (Figure 3.3) appears on your phone, create a script (shown as Example 4) that displays a multi-query dialog that asks you to type your first name into the upper field and your last name into the lower field. After typing your names and pressing the 'OK' button, a popup note should appear, displaying your full name.



Figure 3.3 A multi-query dialog

Example 4: Multi-query dialog

```
import appuifw
names = appuifw.multi_query(u"First name:", u" Last name:")
if names:
    first, last = names
    appuifw.note(u"Your full name is: " + first + " " + last)
else:
    appuifw.note(u"Cancel!")
```

First we import the module `appuifw`. Then we pass two Unicode strings to the `multi_query()` function. The function returns the user's input from the fields into a variable called `names` as a tuple.

In the third line of code we use an `if` statement. If the variable `names` holds some content, that is, the variable `names` is not `None`, then the condition is true and we read the tuple's entries and hand them over to the variables `first` and `last`. The `if` statement is described in the language lesson more thoroughly.

Next we create an informative popup note. The text that we pass to the note function consists of three parts: the string `u"Your full name is: "` and the stored values in the `first` and `last` variables combined with a '+' sign and with `" "` as a space between them.

Python Language Lesson: if statement

An `if` statement is needed to make decisions based on variable values:

```
if x > 0:
    appuifw.note(u"x is positive!")
else:
    appuifw.note(u"x is negative or zero.")
```

The Boolean expression after the word '`if`' is called the condition. If the condition is true (in this case, if the value of the variable x is bigger than 0), then the indented statement in the next line (here, `appuifw.note(u"x is positive!")`) is executed. If the condition is false, the statement in the `else` branch is executed (here, `appuifw.note(u"x is negative or zero.")`). Note that you can reverse the outcome of the condition, by putting the `not` keyword in front of the condition.

The most important comparison operators are ==, !=, < and >, for equality, inequality, less than and greater than. Note that if you want to test whether a variable is not false, zero, `None` or an empty string or list, you can use the following shorthand:

```
if x:
    appuifw.note(u"x is not empty")
```

You can test many conditions in sequence:

```
if x > 0:
    appuifw.note(u"x is positive!")
elif x == 0:
    appuifw.note(u"x is zero.")
else:
    appuifw.note(u"x is negative.")
```

Here, `elif` stands for 'else if'. You can have multiple `elif` statements after each other. They are evaluated in sequence until one of the conditions evaluates to true.

The indented statements that follow a condition are called a block. The first unindented statement marks the end of the block. You must indent the statements in the block (see Section 2.4), otherwise the Python interpreter does not know where the block begins and ends. In other programming languages, for example, C++, this is solved by starting and ending a block with brackets. Having no brackets in Python makes the code easier and clearer to read.

Python Language Lesson: list

A list is an ordered set of values. The values that make up a list are called its elements or items. The items can be of any type, such as strings or integers. Items of the list do not have to be of the same type, so you can have a list that contains both strings and integers.

There are several ways to create a new list. The simplest is to enclose the elements in square brackets:

```
mylist = [u"Symbian", u"PyS60", u"MobileArt"]
```

Each item in the list can be accessed by its index in square brackets after the list name: in the example, `mylist[0]` refers to the value `"Symbian"` and `mylist[2]` to `"MobileArt"`. We can also assign values to specific items in the list:

```
mylist[1] = "Python"
```

You can access parts of the list easily with the `[start:end]` notation for the index – this is called slicing the list. For instance, `mylist[1:2]` returns a new list that includes the items `["Pys60", "MobileArt"]`. Optionally, you can leave out either the start index or the end index. For instance, `mylist[:1]` returns a new list with the items `["Symbian", "PyS60"]`.

To add another element to the end of the list, use the `append()` function. For instance, the following line

```
mylist.append(u"Fun")
```

would change the list to contain the following items: `["Symbian", "PyS60", "MobileArt", "Fun"]`.

3.2.4 Popup Menu: popup_menu

Syntax: `popup_menu(list[, label ])`
Example code:

```
appuifw.popup_menu(choices, u"Select:")
```

The `popup_menu()` function displays a list of items in a popup dialog. You need to give a list of Unicode strings in the parameter `list`. You can also define a `label` which is the text that appears at the top of the list.

This function returns the index of the chosen item in the list. If the user cancels the dialog by pressing Back, `None` is returned.

Exercise

Create a script (see Example 5) that displays a popup menu with three entries, such as the one in Figure 3.4. After selecting any of the choices and pressing 'OK', you should see a note appear with different text depending on which selection you made. Use the phone's up and down keys to make the selection from the menu.



Figure 3.4 A popup menu

Example 5: Popup menu

```
import appuifw
choices = [u"Symbian", u"PyS60", u"MobileArt"]
index = appuifw.popup_menu(choices, u"Select:")
if index == 0:
    appuifw.note(u"Symbian, aha")
elif index == 1:
    appuifw.note(u"PyS60 - yeah")
elif index == 2:
    appuifw.note(u"I love MobileArt")
```

First we import the module `appuifw`. In the next line, we create a list with three elements `[u"Symbian", u"PyS60", u"MobileArt"]`. Each element represents an entry in the menu. The list is assigned to the variable `choices`.

In the next line, we use the `popup_menu()` function and pass the variable `choices` as the first parameter to the function. We also pass a Unicode string `u"Select:"` as the second parameter to the function.

When our script is executed, the function returns an index that refers to one of the elements of our list – depending on the user's selection. The returned index – a number – is stored in a variable that we have named `index`.

Now, we need some code that displays a note dialog with different content depending on the user's selection. This is done using the `if` statement, simply by checking the returned list index and comparing it to all possible options: 0, 1, 2. If the user cancels the dialog, none of the conditions is true and no dialog is shown.

3.2.5 Selection List: selection_list

Syntax: `appuifw.selection_list(choices, search_field)`
Example code:

```
appuifw.selection_list(colors, 1)
```

The `selection_list()` function executes a dialog that allows the user to select a list item. It works similarly to the `popup_menu()` function except that it has no label on the top. Instead, it has an option to display a built-in search field (a find pane) at the bottom (see Figure 3.5).

Compared to the popup menu, the selection list is more suitable for presenting long lists, especially because of the search field that lets you find items in the list character by character. Setting the parameter `search_field` to `1` enables the find pane. It is disabled by default. If the find pane is enabled, it appears only after you press a letter key.

The `selection_list()` function returns the index of the chosen item or `None` if the selection is cancelled by the user.

Exercise

To see how the selection list appears on your phone, create a script (see Example 6) that displays a `selection_list()` dialog with four entries – 'red', 'green' 'blue' and 'brown', as in Figure 3.5.

When running the script, type the letter 'b'. The find pane should appear at the bottom of the list and the entries 'blue' and 'brown' should appear. Select 'blue' and press the OK button. Text should appear, saying

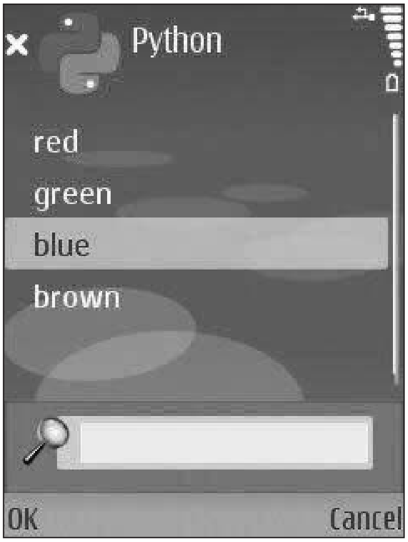


Figure 3.5 Selection list

'blue is correct!'. If you select something else, for instance red, a text should appear stating 'Bzz! red is not correct'.

Example 6: Selection list

```
import appuifw
colors = [u"red", u"green", u"blue", u"brown"]
index = appuifw.selection_list(colors, 1)
if index == 2:
    print "blue is correct!"
elif index != None:
    print "Bzz! " + colors[index] + " is not correct"
```

We create a list with four colors that are specified as Unicode strings. Each element represents an entry in our selection list dialog. The list is assigned to the variable `colors`.

In the next line, we use the `selection_list()` function and pass `colors` to it. We also enable the find pane by setting 1 as the second parameter. When our script is executed – and the user makes a selection – this function returns an index that refers to an element inside our list of colors.

Now, we need some code that determines whether the user has selected the third item in the list or not. We compare the value of the `index` variable with the correct index, 2. If the condition is true, we print the text 'blue is correct!' to the screen. Otherwise, if the user did not cancel the dialog, our script enters the `elif` block.

There, we combine another string to be printed out on the screen. We pick the chosen color name from the `colors` list using the returned index in the variable `index` and put it in the middle of the string to be printed out.

In Python, you can print out almost any value, including lists and tuples, and see the contents on the screen in a human-readable format. This makes the print statement especially useful for debugging. If you want to know more about debugging at this point, you should look at Appendix B C where we explain the easiest ways to debug PyS60 programs.

Python Language Lesson: print statement

The print statement causes the PyS60 interpreter to display a value on the screen:

```
print "hello"
```

There is no need to indicate that it is a Unicode string because it uses only the resources of the Python interpreter and no native resources from the S60 platform.

3.2.6 Multi-Selection List: multi_selection_list

Syntax: `multi_selection_list(choices[, style, search_field ])`
Example code:

```
appuifw.multi_selection_list(choices, 'checkbox', 1)
```

The `multi_selection_list()` function shows a dialog that allows the user to select multiple list items. The function returns a tuple of indices of the chosen items, or an empty tuple if the user cancels the selection.

In Example 7, the parameter `choices` is a list of Unicode strings and `style` is an optional string with the default value `"checkbox"`. This dialog shows a check box to the left of each item in the list. You can browse the list with the navigation keys on the mobile phone keyboard and select items by pressing the Select key (see Figure 3.6).

The `search_field` parameter is 0 (disabled) by default and it is optional. Setting it to 1 shows a find pane that helps the user search for items in long lists. If enabled, the find pane is always visible with the list.

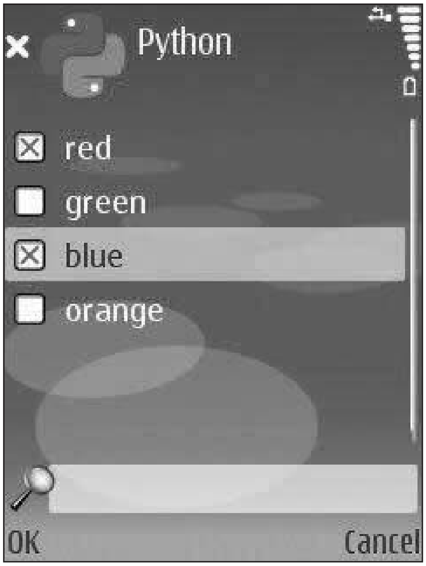


Figure 3.6 Multi-selection list

Example 7: Multi-selection list

```
import appuifw
colors = [u"red", u"green", u"blue", u"orange"]
selections = appuifw.multi_selection_list(colors,'checkbox',1)
print selections
```

Python Language Lesson: for loop

The `for` loop goes through the items of a list. In the example below, we define a list of foods. Then we loop through the food items one by one, printing each of them to the screen.

```
foods = [u"cheese", u"sausage", u"milk", u"banana", u"bread"]
for x in foods:
    print x
```

This almost reads like English: 'For x (every element) in (the list of) foods, print (the name of) x'. When executing this for loop, the resulting screen shows 'cheese', 'sausage', 'milk', 'banana' and 'bread', one item per line.

If you want to loop over a range of numbers, you can use a special `range()` function. This function produces a list of integers, from 0 to the given maximum value.

```
for i in range(10):
    print i
```

This prints out the numbers from 0 to 9.

Python Language Lesson: while loop and break

The `for` loop is handy if you have a list of items or you know the maximum number of iterations beforehand. Sometimes, however, you want to keep looping until some condition becomes true. The `while` loop comes in useful in this case. This is how it looks:

```
i = 10
while i > 5:
    print x
    i = i - 1
```

This example prints out 10, 9, 8, 7, 6 on the screen. After the word `while`, a conditional expression follows just as in an `if` statement.

Another typical use is as follows:

```
while True:
    ret = appuifw.query(u"Continue?", "query")
    if not ret:
        break
```

Here, the condition is always true, so the loop goes on forever unless we force the execution out of it using the `break` statement. The break statement makes the execution jump out of the loop instantly. In this example, we show a query dialog in each iteration. The looping continues until the user cancels the dialog.

3.3 Messages

Let's introduce a second PyS60 module: `messaging`. It handles the sending of SMS (Short Message Service) and MMS (Multimedia Messaging Service) messages. Here we use only one function from the module: `messaging.sms_send()`.

```
import messaging
messaging.sms_send("+14874323981", u"Greetings from PyS60")
```

To be able to send an SMS we need to import the module `messaging` at the beginning of our script. We only need to pass two parameters to the function `sms_send` (): the telephone number of the recipient as a string and the body of the message as a Unicode string.

In this concluding example (Example 8), we combine the UI elements learned in this chapter and the function for sending SMS messages into a useful little program.

Imagine you are at home, your fridge is empty and your friend is on the way home passing a grocery store. He does not know which items to buy. You want to send him a list of items by SMS while standing in front of the fridge checking what is needed. Typing all the stuff into the phone is just too much hassle and takes too much time.

This is where your PyS60 program, the Shopping List Assistant, comes in. From a pre-defined list of items on your screen, you can select the items needed using a `selection_list()` dialog. A text input field lets you type an additional greeting. After this, your shopping list is on its way to your friend by SMS.

Example 8: Shopping list assistant

```
import appuifw, messaging
foods = [u"cheese", u"sausage", u"milk", u"banana", u"bread"]
choices = appuifw.multi_selection_list(foods,'checkbox',1)
shoppinglist = ""
for x in choices:
    shoppinglist += foods[x] + " "
greetings = appuifw.query(u"Add some greetings?", "text", u"thanks!")
if greetings:
    shoppinglist += greetings
print "Sending SMS: " + shoppinglist
messaging.sms_send("+1234567", shoppinglist)
appuifw.note(u"Shopping list sent")
```

First we import the modules `appuifw` and `messaging`. Next we create the list `foods`, which includes every possible food item that can be missing from the fridge. This list can be extended by you to as many elements as you want. Remember that the elements need to be Unicode strings, that is, they must be prefixed with the ‘u’ character.

In the next line, we generate our `multi_selection_list()` and pass `foods` to it. We also enable the find pane. This displays the selection list on the phone screen with all the goods, allowing us to select all the items that need to be bought. The result of our selection is stored in the variable `choices`. It is a tuple that holds the indices of the selected items.

A `for` loop reads the selected items from the `foods` list, based on the indices of the selected items in `choices`. We start with an empty list, `shoppinglist`, to which we add the required items during each

iteration of the `for` loop. The += operator is a shorthand for the following expression:

```
shoppinglist = shoppinglist + foods[x] + " "
```

In the line following the `for` loop, we create a query dialog of type 'text', meaning that the phone screen will show a text input field saying 'Add some greetings?' with the initial word 'thanks!'. If the user chooses to add a greeting, the string is added to the shopping list.

Finally, we want to send the shopping list by SMS. For this we use the `messaging.sms_send()` function to which we hand the mobile phone number of the recipient and the list `shoppinglist`.

Once the phone has sent the SMS, we want a confirmation that the SMS with our shopping list has been sent. We do this by generating a note dialog of type 'info'.

And that is all. Let's hope our fridge will never be totally empty again.

3.4 Summary

In this chapter, we have introduced a set of native UI dialogs that PyS60 offers. They got you involved with working examples of PyS60 programming with relatively little code to write. The final exercise showed how to combine several elements into a simple but useful application, the shopping list assistant.

In Chapter 4, we introduce principles of application building and move into more fun applications. In many examples throughout the book, you will find these UI elements again and see how they can be part of a fully working application. At the same time, you learn how you can integrate them into applications of your own.

Maybe you already have an idea of a simple application in which some of these UI elements can be used. Why not try it out right away?

4

Application Building and SMS Inbox

Python for S60 gives you easy access to the framework for building user interfaces on the S60 platform. Thanks to PyS60's simple way of handling this, you learn how to build a real application that includes a title bar, a menu and some dialogs in around 15 lines of code!

We start by describing how functions are defined in Python. We build an application in Section 4.2, which also presents an overview of the structure of a typical S60 application. In addition, this section introduces one of the most important concepts in PyS60, callback functions, which let you bind arbitrary actions to various events. This concept comes in useful when we show you how to add menus to your application.

You may already be eager to start making real applications. In Section 4.3, we show how to handle strings. This is put to good use right away when we explain how to access the SMS inbox in Section 4.4. For example, we build fully functional applications which let you search and sort your SMS messages easily. Finally, in Section 4.5, we present an SMS game server that lets you and your friends play the game of Hangman using text messages.

4.1 Functions

The Python scripts that you tried in the previous chapter are executed line by line. Each line of code performs one action and, after the action is finished, execution moves to the next line. This is a simple and robust way to perform tasks that straightforwardly proceed from the beginning to the end.

However, if we want the user to decide what to do, instead of letting the program always perform the same operations in the same order, the code must be structured differently. Typically, this is the case when

an application has a graphical user interface that lets the user perform different actions by interacting with user interface (UI) elements.

When using the S60's framework for building user interfaces, the execution does not progress deterministically line after line in the code. Instead, the user may launch specific tasks by pressing, say, specific keys on the mobile phone keyboard. In this case, your job as an application developer is to bind specific tasks to specific key events. When the user chooses a menu item or to quit, your application should execute a task that corresponds to such an event. For this purpose, you need to make functions of your own.

In Example 1 (shown here once more as Example 9), we had three lines of code that were executed in sequence. The second line triggered a text input field and the third line a popup note.

Example 9: First PyS60 program

```
import appuifw
word = appuifw.query(u"Type your name", "text")
appuifw.note(u"Greetings from " + str(word))
```

We could define a function and put the two lines of code inside the function. This is what we do in Example 10, where the name of the function is `askword()`. Within a function, the execution proceeds sequentially as usual. Functions are just a way to divide and structure code into small chunks that are given a name.

Example 10: First function

```
import appuifw
def askword():
    word = appuifw.query(u"Type a word", "text")
    appuifw.note(u"The word was: " + str(word))
```

Any time we want to carry out this task, that is, to execute these two lines of code, we just call the function `askword()`:

```
askword()
askword()
```

In this case, it asks for a word twice.

You have been calling functions all the time in the previous chapters. For example:

```
messaging.sms_send("+358...", u"Greetings from PyS60")
appuifw.note(u"I love MobileArt")
```

These lines from earlier examples, call the function `sms_send` in the `messaging` module and the function `note` in the `appuifw` module.

Python Language Lesson: function

Functions are a way to divide your code into independent tasks. A function has a name and a body and it may be called. Optionally, a function may take input variables, which are called parameters, and it may return an output value. Here is an example:

```
def add_values(x, y):
    print "Values are", x, y
    return x + y
```

The keyword `def` starts the function definition. After the keyword `def` comes the function name, in this case `add_values`. In parentheses, follow the function parameters separated by commas. A colon ends the function header.

The indented lines below the function header form the function body, which is executed line by line when the function is called. In Python, indentation makes a difference, so make sure that the lines are aligned correctly in the function body. Function bodies follow the same indentation rules as `if` clauses and loops. The function body may contain a `return` statement followed by a value which is returned to the calling line. The `return` statement is optional.

A function is called with the function name followed by the input values in parentheses:

```
z = 3
new_sum = add_values(z, 5)
print "Their sum is", new_sum
```

If no input values are given, there is nothing between the parentheses.

The return value may be assigned to a variable, for example `new_sum = add_values(z, 5)`, or the function call may be included in a more complex expression. For example, the code of the last line in the example above could have read:

```
print "Their sum is", add_values(z, 5)
```

In this case, you would not need the `new_sum` variable in the example above. As you might guess, the example produces this output:

```
Values are 3 5
Their sum is 8
```

Python's slogan is 'Batteries included'. This refers to the fact that Python comes with a comprehensive library of pre-made functions which make many complex tasks practically effortless.

4.2 Application Structure

Many S60 applications share the same user interface layout. Take a look at Figure 4.1 or almost any application on your S60 mobile phone, including the PyS60 interpreter and you will notice the same structure in the user interface.

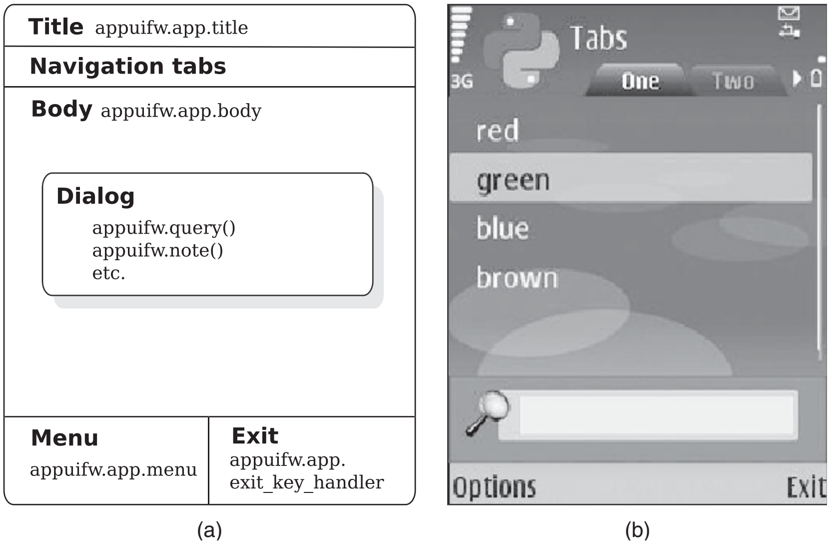


Figure 4.1 A typical user interface (a) structure and (b) screenshot based on the structure

Figure 4.1(a) shows the structure of the S60 user interface. To see how the diagram maps to reality, compare it to Figure 4.1(b), which shows a typical user interface that is built using the S60 UI framework. In the following description, text in parentheses refers to text in the screenshot.

At the top of the screen, you see the application title (Tabs). Below the title, you may see a row of navigation tabs (One and Two). The large area in the middle is the application body which may be used by a number of different UI elements (the list: red, green, blue, brown). Various dialogs, such as popup notes, may appear on the application body, as we saw in Chapter 3.

At the bottom, you see two items that are activated by two dedicated keys, the left and right softkeys, on your mobile phone keyboard. If no

dialog is active, the left softkey activates the application menu (Options) and the right softkey quits the running application (Exit). If a dialog is shown, the left softkey corresponds to Accept and the right one to Cancel.

In PyS60, you can access the UI elements through a special `app` object that is part of the `appuifw` module. We talk more about objects in Section 4.2. Modifying the UI elements is easy: each of the elements (`title`, `body`, `menu` and `exit_key_handler`) is a special variable inside the `appuifw.app` object and you can assign values to them as you can to any other variable. Just remember to use the full name, for instance `appuifw.app.title`. Whenever you change any of these variables, the corresponding UI element changes accordingly.

The first application is a minimalist application that uses the UI framework provided by the `appuifw` module. It does not do anything useful but it illustrates the minimum requirements for a working application. It runs until the user chooses to quit the application, as opposed to previous examples, which executed deterministically from the beginning to the end. Figure 4.2 shows it in action.



Figure 4.2 First application

This section includes two language lessons that are related to application building, about callback functions and objects. Do not worry if you cannot grasp them at once. They will become clear with the many more examples that follow.

Example 11: First application

```
import appuifw, e32
def quit():
    print "Exit key pressed!"
    app_lock.signal()

appuifw.app.exit_key_handler = quit
appuifw.app.title = u"First App!"
appuifw.note(u"Application is now running")

app_lock = e32.Ao_lock()
app_lock.wait()
print "Application exits"
```

Besides importing the familiar `appuifw` module, we also need a module named `e32`. This module offers many useful low-level utility objects and functions related to Symbian OS functionalities. Here we need the object `e32.Ao_lock()`.

We define a new function, `quit()`, that takes care of shutting down the application when the user presses the Exit key. Since we could have a number of functions to perform various tasks, we need to tell Python which function is dedicated to handling the Exit events.

This is done by assigning the function name (not its value) to the special `appuifw.app.exit_key_handler` variable. When the user presses the Exit key, the function that this variable refers to is called by the UI framework. In many situations, Python expects you to provide a function name that is used to call the corresponding function when some event has occurred. Functions of this kind are called *callback functions*. The language lesson about callback functions clarifies the concept.

Python Language Lesson: callback function

A callback function is an ordinary function, defined similarly to any other function, but it is used for a specific purpose. There is no technical difference between ordinary functions and callback functions. The distinction is made to clarify discussion.

Typically, a callback function is called by a function in the PyS60 library to respond to a specific event – such as when the user has chosen a menu item or decides to quit the application. In contrast, ordinary functions are called in your application code to handle application-specific tasks. However, in some cases a callback function may be called by your application explicitly.

Associating a function with an event is often called *binding*. Some PyS60 objects, such as `Canvas` and `Inbox`, include a function called `bind()` that is used to bind a callback function to some event related to the object.

Whenever the PyS60 API documentation or this book asks you to provide a callback function, do not add parentheses after the function name since the function is only called after the event occurs. If you are familiar with C or C++, you might notice that this is similar to how function pointers are passed around in these languages.

In Example 11, we then assign the application name to the title variable `appuifw.app.title`. This string shows at the top of the screen when the application is running.

As we discussed at the beginning of this chapter, the PyS60 examples we have seen so far are executed line by line and they exit when the last line has been executed. However, Example 11 should not exit until the user decides to do so. Therefore, our application should not drop out after the last line has been executed, but instead should wait for user action.

This is accomplished with an object called `Ao_lock` that is part of module `e32`. The object includes a function called `wait()` that puts the application into a waiting mode until the lock is explicitly released with the `signal()` function. The `signal()` function must be called when you want to terminate the application, so we call it inside the `quit` function.

To see why it's important to have the lock at the end of the application code, omit the line, `app_lock.wait()`, from your code and run the application. A dialog pops up but the application does not go into a wait state; it finishes instantly.

When you run Example 11, you should see the 'First App!' title at the top of the screen (as in Figure 4.2) and a dialog popping up. Nothing else happens until you press the Exit key (the right softkey), which shuts down the application. After this, you should see the text 'Exit key pressed!' and 'Application exits' messages in the PyS60 console on the phone screen.

It might be difficult to notice that the application is running at all. This is because the PyS60 interpreter is built using exactly the same application framework, so it looks the same as any other PyS60 application.

Python Language Lesson: object

Objects hold together variables and the functions that manipulate them. In many cases, functions and variables are so closely related to each other that they would be meaningless if they were handled separately. Objects are especially useful in large, complicated applications which would be practically incomprehensible if they were not divided into smaller units.

Python does much of this work for you, so you can code happily with Python's ready-made objects. Developing large and complex applications in an object-oriented manner is out of the scope of this

book. There is a lot of information about object-oriented programming in other books and on the web. You may be surprised to see that example applications which are used to teach object orientation are not complex at all with Python and they do not require you to derive objects of your own.

For our purposes, it is enough to know that some functions create objects. In Example 11, an `Ao_lock` object is assigned to the variable `app_lock`.

```
app_lock = e32.Ao_lock()
```

The object itself contains variables and functions that can be accessed with the dot notation, for example:

```
app_lock.wait()
```

calls the object's `wait()` function. Likewise:

```
appuifw.app.title = u"Hello World"
```

assigns a value to the `title` variable of the `app` object in the `appuifw` module. In practice, you can treat objects as if they are modules inside modules, as the dot notation suggests.

In this chapter, you can find many examples of object usage. You have already seen the `app` object which handles the standard user interface and the `Ao_lock` object which is used to wait for user actions. In practice, everything in Python is an object, including strings, lists and tuples, as well as more specific constructs such as the `Inbox` object, which is used to access SMS messages. Objects are so ubiquitous in Python that you use them all the time without even noticing it.

4.2.1 Application Menu

The application menu provides an easy and familiar way to present a list of available operations to the user. After reading this section, you can start building useful tools with an application menu, which perform tasks according to the user's requests.

Python Language Lesson: tuple

A tuple is an immutable list. You can't add, remove or modify its values after creating it. Tuples are often used to group related values, such as

coordinates, address lines or ingredients of a recipe. Tuples are defined in the same way as lists but, instead of square brackets, values are enclosed in parentheses:

```
red = (255, 0, 0)
weekdays = ("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
family = (("Mary", 38), ("John", 35), ("Josephine", 12))
```

The first and second lines define simple sequences. The last line makes a tuple of tuples. The inner tuples contain different types of value: strings and integers.

You can easily *unpack* values from tuples back to individual variables. Here we use the tuples `red` and `family` that were created above:

```
r, g, b = red
mom, dad, child = family
```

In this case, the variables `r`, `g` and `b` are integers and `mom`, `dad` and `child` are two-item tuples. Note that you have to define as many variables in unpacking as there are items in the corresponding tuple.

You may refer to individual items in a tuple with indices, as with lists. In contrast to lists, assignments such as

```
family[1] = ("Jack", 25)
```

are not possible since this would modify the tuple.

In some cases you need to convert lists to tuples or tuples to lists. The former is possible with function `tuple()` and the latter with function `list()`. For example:

```
print list(red)
```

puts out a list to your screen:

```
[255, 0, 0]
```

Note the square brackets that denote a list.

Example 12 extends Example 11 to include an application menu (see Figure 4.3).



Figure 4.3 Application menu

Example 12: Application menu

```
import appuifw, e32

def photo():
    appuifw.note(u"Cheese!")

def darken():
    appuifw.note(u"I can't see a thing!")

def lighten():
    appuifw.note(u"My eyes are burning!")

def quit():
    print "WANNABE PHOTOEDITOR EXITS"
    app_lock.signal()

appuifw.app.exit_key_handler = quit
appuifw.app.title = u"PhotoEditor"
appuifw.app.menu = [(u"Take Photo", photo), (u"Edit photo",
                   ((u"Darken", darken), (u"Lighten", lighten)))]

print "WANNABE PHOTOEDITOR STARTED"
app_lock = e32.Ao_lock()
app_lock.wait()
```

The application menu, `appuifw.app.menu`, is defined as a list that consists of tuples. Each item in the list is of the form `(u"Item name",`

item_handler), where the first element defines the text that is shown in the application menu and the second is a callback function that is called when the item is selected. You can see this more easily by comparing Figure 4.3 with the appuifw.app.menu list.

Menus may be hierarchical, that is, a menu item may open a sub-menu. Sub-menus are defined similarly to the main menus as a list of tuples. In Example 12, the second menu item 'Edit photo' opens a sub-menu of two items, 'Darken' and 'Lighten'. In Figure 4.3, 'Take Photo' and 'Edit Photo' are shown as the main menu and the items 'Darken' and 'Lighten' as a sub-menu corresponding to the 'Edit photo' item.

The three new functions, photo(), darken() and lighten(), are callback functions that bind the actions to the menu items. In this case, each item opens a popup note that relates to the chosen item. As before, you may close the application with the right softkey.

Now you should try to build some menus and simple applications by yourself! Building applications is easy, in practice.

4.2.2 Application Body

There are several possible objects that can be assigned to the application body:

- a canvas that handles graphics on screen (see Chapter 5)
- a form which is used to build complex forms that include combinations of various input fields, such as text, numbers and lists (see Chapter 9)
- a listbox that shows a list of items (see Chapter 9)
- a text object that handles free-form text input (see the PyS60 documentation for more information about this object).

You may increase the area that is reserved for the application body using the appuifw.app.screen variable. Three different sizes are provided (see Figure 4.4):

```
appuifw.app.screen = "full"
appuifw.app.screen = "large"
appuifw.app.screen = "normal"
```

4.2.3 Tabs

In the PyS60 documentation, you can find a description of the appuifw.app.set_tabs() function, which, unsurprisingly, is used to define tabs for the navigation bar. Tabs are defined using a list of strings and callback functions, in a similar way to the application menu.

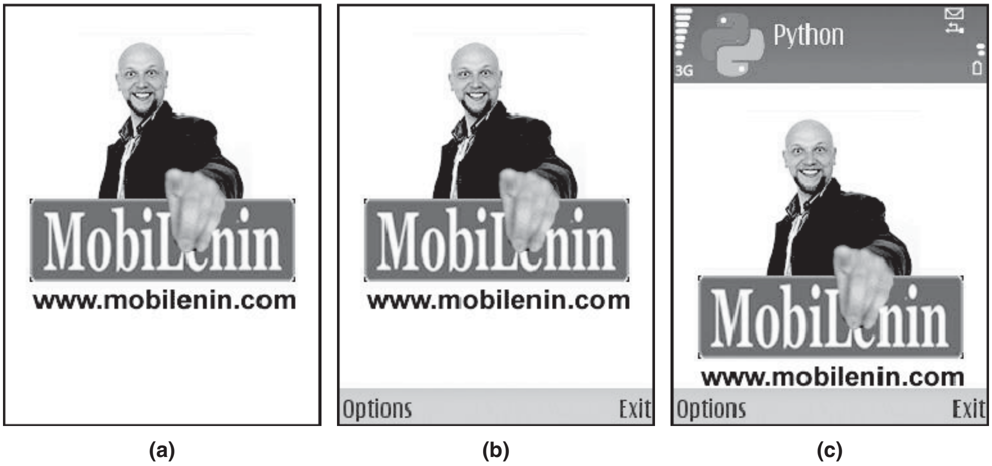


Figure 4.4 Application screen sizes: (a) full, (b) large and (c) normal

4.2.4 Content Handler

A special `Content_handler` object, which is part of the `appuifw` module, is available for opening files of various types, such as images, photos and web pages, using the standard viewer applications of the S60 platform. This object is also used in Chapter 9.

4.3 String Handling

There are only a few applications which do not handle textual data at all. In some cases you are not interested in text as such, but you need to convert textual input to a format which is easy to handle programmatically, for instance to lists or integers. Finally, when your application has processed the data, it is often printed out in textual form, so the application must convert internal data structures back to text.

Luckily, all this is straightforward in Python. In this section we introduce basic operations on strings, which are used by many examples in this book. Section 4.4, which introduces you to the SMS inbox, requires some power tools for string handling.

In Python, strings are objects too. In effect, this means that all string-related functions are available automatically without any imports, but you need to remember to use the dot notation, for example `txt.find("is")`, when calling the string functions. Note that it is not possible to modify a string after creation. All string operations return a new string and leave the original string intact.

4.3.1 Defining a String

There are many ways to define new strings:

```
txt = "Episode XXXIV"
txt = 'Bart said: "Eat my shorts"'
txt = """Homer's face turned red. He replied: "Why you little!""""
```

The first line uses double quotes to enclose a string. This is the most usual way to define a string. The second line uses single quotes. It comes in handy if your string contains double quotes. The last line shows the last resort – if the string contains both single and double quotes and possibly line breaks, you can enclose the string in three sets of double quotes.

4.3.2 Accessing Parts of a String

You can access individual characters with an index enclosed in square brackets, in a similar way to lists. If you want to find the starting index of a specific substring, use the function `find()`. For example, the following code outputs first 'h' and then 'i':

```
txt = "python is great"
print txt[3]
print txt[txt.find("is")]
```

Above, `txt.find("is")` returns the value 7 and the character at index 7 is 'i'.

You can access substrings with the `[start:end]` notation:

```
txt = "python is great"
print txt[:6]
print txt[7:9]
print txt[10:]
```

The print statements output substrings 'python ', 'is' and 'great'.

4.3.3 Making Decisions with a String

Often you need to make a decision based on a string. You can test whether a string is non-empty as follows:

```
txt = ""
if txt:
    print "Many characters!"
else:
    print "No characters"
```

This example prints out 'No characters'. You can find out the length of a string with the function `len()`. The following example outputs 'password too short'.

```
passwd = "secret"
```

```
if len(passwd) < 8:
    print "password too short"
```

If you need to know whether a string contains a specific substring, use the `find()` function. It returns −1 if the substring cannot be found.

```
url = "http://www.google.com"
if url.find(".com") == -1:
    print "Access denied!"
```

If the substring should appear at the beginning of the string, use the `startswith()` function:

```
if url.startswith("http://"):
    print "This is a valid URL"
```

4.3.4 Input Sanitization

Often you need to normalize the user's input so that all input is either in lower or upper case. Functions `upper()` and `lower()` do the job:

```
a = "PyTHon"
print a.upper()
print a.lower()
```

The print statements output 'PYTHON' and then 'python'.

If you need to be sure that a string does not contain any leading or trailing white space, use the `strip()` function. The following code outputs the string 'fancy message':

```
a = " fancy message "
print a.strip()
```

This is actually one of the most often needed operations for strings – in this book it is used in 12 examples. It is especially useful when you receive data from a file or over a network connection.

You can replace a substring in a string with the function `replace()`. The following example outputs 'Python is the future':

```
a = "Java is the future"
print a.replace("Java", "Python")
```

You can also replace individual characters in a string, which is often useful for cleaning up input, say, from a web service. The following example removes all spaces in a string and outputs '1,2,3,4,5,6'.

```
a = "  1, 2,  3, 4,  5,6 "
print a.replace(" ", "")
```

If you need to cut a string into a list of substrings, use the function `split()`. In default mode, `split()` cuts the string at every space character. If you give a string parameter to `split()`, it is used as the delimiter instead. The following example prints out the list `["one", "two", "three", "four"]`.

```
txt = "one:two:three:four"
print txt.split(":")
```

4.3.5 Formatting Output

The real workhorse of Python's string handling is the string-templating mechanism. It is used to assemble a string based on other variables, which can be integers, floating-point numbers or other strings. To insert given values in the middle of a string, you need to use special placeholders that start with the character '%'.

The placeholders are replaced with variable values which are given in a tuple. The template string and the corresponding input tuple are combined with a percent sign operator, as shown below.

The placeholders must agree with the variable types, so if your variable contains an integer, you must use placeholder '%d'. If your variable is another string, the corresponding placeholder should be '%s'. A comprehensive list of possible placeholders and their modifiers can be found in the Python documentation.

Here is an example:

```
print "There are only %d lines in my code" % (5)
print "Hello %s of %d" % ("world", 2007)
ans = [45, 32, 12]
print "the correct answers are %d, %d, and %d" % tuple(ans)
```

You should guess the output. Note that on the last line the input is given as a list which is converted to a tuple by the function `tuple()` before it is used to fill in the placeholders.

You can use the '%d' placeholder to convert an integer to a string. To convert a string to the corresponding integer, use function `int()`. For example, `int("12")` produces integer 12, but `int("a12")` fails since the input contains an invalid value.

Exercise

To get some practice with the string operations, let's consider the following scenario. There is a song contest running on television and the audience can vote for their favorite song and singer by SMS.

There are four songs which you are interested in: 1, 3, 9 and 11. The competition is such that each song can be sung by any singer. Your favorite singers are Tori, Kate and Alanis. To maximize the probability that any of your favorite singers will get to perform at least one of the songs, you decide to vote for all the singer–song combinations. Since you realize that writing 12 text messages of format 'VOTE [song] [singer]' manually is too much work, you quickly code the script in Example 13 to do the job.

Example 13: SMS voter

```
import messaging

PHONE_NUMBER = "+10987654321"
songs = (1, 3, 9, 11)
singers = "Tori,  Kate,  Alanis"

for song in songs:
    for singer in singers.split(","):
        msg = u"VOTE %d %s" % (song, singer.strip())
        print "Sending SMS", msg
        messaging.sms_send(PHONE_NUMBER, msg)
```

Here, the songs of interest are stored in the `songs` tuple. Your favorite singers are given in the comma-separated string `singers`. Since we want to go through every possible singer–song combination, we first start looping through all `songs` and for each `song` we loop through all `singers`.

However, we need to convert the `singers` string to a list for looping, so we splice the string using the `split()` function. The `split()` function produces the list `["Tori", " Kate", " Alanis"]`. Note that the names of Kate and Alanis are preceded by an unnecessary space.

We use the function `strip()` to get rid of leading white space. Finally we construct a voting message with a template string. Since the song identifiers are integers, they require the `%d` placeholder. Likewise, the string placeholder `%s` is used for the names.

Be careful with the combination of loops and the `sms_send()` function, such as the one above. In real life, each SMS message costs money and with loops you can easily send thousands of them! You might want to try out this example without a SIM card in your phone, just to be on the safe side.

4.4 SMS Inbox

Even though a large number of today's mobile phones support various Internet protocols, from VoIP and instant messaging to HTTP, only SMS

can guarantee smooth interoperability between all devices, including old models. It is reasonable to assume that every mobile phone user knows how to send and receive SMS messages. Because of this, SMS messages are still widely used to interact with various services, from buying bus tickets to television shows and online dating. Let your imagination free and come up with novel ways to use these services with Python!

You have already seen how to send SMS messages with the `sms_send()` function. In addition, PyS60 provides the `Inbox` object, part of the `inbox` module, that can be used to access messages in your phone's SMS inbox. The latest versions of PyS60 may include access to other SMS folders as well, such as the outbox for sent messages.

You can also define a callback function that is called when a new message is received. This powerful feature enables you to build SMS services of your own, or you can turn your phone into an SMS gateway that forwards received messages to a new phone number.

4.4.1 Accessing the SMS Inbox

Let's start with an example which shows five messages from your SMS inbox.

Example 14: SMS inbox

```
import inbox, appuifw
box = inbox.Inbox()
for sms_id in box.sms_messages()[:5]:
    msg = box.content(sms_id)
    appuifw.note(msg)
```

Besides the familiar `appuifw` module, which is needed for showing notes, we need to import the `inbox` module that encapsulates access to the SMS inbox.

All functions related to receiving SMS messages belong to the `Inbox` object which is created with the `inbox.Inbox()` function. To read an individual SMS message, you need to know its message ID.

The `Inbox` object's function `sms_messages()` returns message IDs for all messages in your phone's SMS inbox. Since you may be a texting maniac who keeps thousands of messages in the inbox, this example shows up to five of them. From the list returned by the `sms_messages()` function, we slice off the first five items using the slicing operator `[:5]`.

The `content()` function returns the content of a message given its message ID. The content, or the actual text of the message, is then shown in a popup note.

4.4.2 SMS Search

Example 15 is a useful script: a search tool for your SMS inbox. First the script asks for some text for which to search. Then it goes through your SMS inbox and shows excerpts of messages which contain the given string. You can see the full message by selecting a desired list item.

Example 15: Inbox search

```
import inbox, appuifw
box = inbox.Inbox()
query = appuifw.query(u"Search for:", "text").lower()
hits = []
ids = []
for sms_id in box.sms_messages():
    msg = box.content(sms_id).lower()
    if msg.find(query) != -1:
        hits.append(msg[:25])
        ids.append(sms_id)

index = appuifw.selection_list(hits, 1)
if index >= 0:
    appuifw.note(box.content(ids[index]))
```

First, the string for which to search is stored in the variable `query`. As in the previous example, we use the `sms_messages()` function of the `Inbox` object to retrieve a list of all message IDs.

The program loops through all messages, retrieves the contents of each message with the `content()` function and tries to find the `query` string in each message. If the `find()` function reports that the `query` string occurs in the message, we add the message ID to the list `ids` so that it can be found later. Also, we add an excerpt (the first 25 characters) of the message contents to the list `hits`, so that we can present this match to the user.

When we have processed all messages, we show the hits in a selection list using the message excerpts as list items. When the user chooses an item, the list dialog returns the item's index in the list. Using this index, we find the corresponding message ID in the `ids` list and the full message content can be fetched from the phone's SMS inbox and shown to the user.

4.4.3 SMS Sorter

Besides the message content, it is possible to fetch the timestamp, the sender's phone number (address) and information on whether the message has been read already using the `Inbox` object.

To show how to use these functions, we make an application that includes an Exit key handler, an application title and a menu. This

application can be used to sort your SMS inbox according to various attributes of text messages. You can choose, using the application menu, whether the messages should be sorted by time, sender or unread flag.

Example 16: Inbox sorter

```
import inbox, appuifw, e32

def show_list(msgs):
    msgs.sort()
    items = []
    for msg in msgs:
        items.append(msg[1][:15])
    appuifw.selection_list(items)

def sort_time():
    msgs = []
    for sms_id in box.sms_messages():
        msgs.append((-box.time(sms_id), box.content(sms_id)))
    show_list(msgs)

def sort_sender():
    msgs = []
    for sms_id in box.sms_messages():
        msgs.append((box.address(sms_id), box.content(sms_id)))
    show_list(msgs)

def sort_unread():
    msgs = []
    for sms_id in box.sms_messages():
        msgs.append((-box.unread(sms_id), box.content(sms_id)))
    show_list(msgs)

def quit():
    print "INBOX SORTER EXITS"
    app_lock.signal()

box = inbox.Inbox()
appuifw.app.exit_key_handler = quit
appuifw.app.title = u"Inbox Sorter"
appuifw.app.menu = [(u"Sort by time", sort_time),
                    (u"Sort by sender", sort_sender),
                    (u"Unread first", sort_unread)]

print "INBOX SORTER STARTED"
app_lock = e32.Ao_lock()
app_lock.wait()
```

The code may seem long but notice that the functions `sort_time()`, `sort_sender()` and `sort_unread()` are almost identical. Each of these functions creates a list, `msgs`. In each case, the list contains tuples in which the first item is the key, according to which the messages should be sorted, and the second item is the message content.

We use the list's `sort()` function to sort the list. In Python, lists are sorted as follows: if a list contains only strings, they are sorted into alphabetical order; if a list contains only integers, they are sorted into ascending order; if a list contains tuples or lists, the first item of the list or tuple is used as the sorting key; if the list contains items of different types, items of the same type are grouped together and each type is sorted separately.

In this case, the sorting key depends on the user's choice:

- The `sort_time()` function uses the `Inbox` object's `time()` function to fetch the time when the message was received. The time is returned in seconds since 1 January 1970, which is the standard starting point for time measurements in many operating systems.
- The `sort_sender()` function uses the sender's phone number, returned by the `address()` function, as the sorting key.
- The `sort_unread()` function uses the function `unread()` which returns 1 if the message is unread and 0 otherwise.

Since Python's `sort()` function sorts integers into ascending order, we have to make the timestamp and the unread flag negative, `-box.time(sms_id)` and `-box.unread(sms_id)`, as we prefer the newest and the unread messages at the top of the list.

Each of these three functions prepares a list, `msgs`, that is ready to be sorted in the desired way. The actual sorting and showing the resulting list is the same in all the cases, so this task is separated into a function of its own, `show_list()`.

After the `msgs` list has been sorted, we can drop off the sorting keys and use only the second item in the tuple, the message content, to construct a new list that includes message excerpts, as in the previous example. After this, the sorted messages are ready to be displayed.

You could integrate Examples 15 and 16 to make a combined `Inbox` searcher and sorter – a fully fledged personal messaging assistant. There is one more related function, `delete()`, available in the `Inbox` object. It works exactly like the functions `content()`, `address()` and `time()`. It is used to delete a message, so make sure you have backed up your SMS inbox before trying it!

4.4.4 Receiving Messages

If you want to build a service which is controlled by SMS messages, your program must be able to react to incoming messages. In theory, you could accomplish this task by calling the `sms_messages()` function at constant intervals and checking whether the list has been changed since the last call.

This approach is likely to perform lots of unnecessary work. Luckily, PyS60 provides an alternative: a callback function can be called whenever a new message is received.

Example 17: SMS receiver

```
import inbox, appuifw, e32

def message_received(msg_id):
    box = inbox.Inbox()
    appuifw.note(u"New message: %s" % box.content(msg_id))
    app_lock.signal()

box = inbox.Inbox()
box.bind(message_received)

print "Waiting for new SMS messages.."
app_lock = e32.Ao_lock()
app_lock.wait()
print "Message handled!"
```

The Inbox object's `bind()` function binds a callback function to an event that is generated by an incoming message. In this example, the function `message_received()` is called when a new message arrives.

The callback function takes one parameter, namely the ID of the incoming message, `msg_id`. Based on this ID, the message contents, or any other attribute, can be retrieved from the SMS inbox, as we saw above.

The program should not execute from start to finish at once. Instead, it should wait for an event to occur. This time the event is not generated by the user directly, but by an incoming SMS message. Waiting is handled in the familiar manner, using the `Ao_lock` object.

When a new message arrives, the function `message_received()` is called. It opens a popup note that shows the contents of the newly arrived message. Then the `Ao_lock` is released and the program finishes with the message 'Message handled!'. The only way to terminate this program properly is to send an SMS message to your phone.

4.4.5 Forwarding Messages

As you know from everyday usage of your mobile phone, an arriving text message triggers a sound notification, as well as a popup note saying that a new message has arrived.

However, if you make a program that handles messages automatically, the notification may not be desirable. Luckily, the sound and the dialog can be avoided if the message is deleted immediately in the callback function when the new message has arrived.

In Example 18, we demonstrate this feature with a filtering SMS gateway. The gateway receives a new message, removes all nasty words in it and forwards the censored message to another phone.

Example 18: Filtering SMS gateway

```
import messaging, inbox, e32

PHONE_NUMBER = "+18982111111"
nasty_words = ["Java", "C++", "Perl"]

def message_received(msg_id):
    box = inbox.Inbox()
    msg = box.content(msg_id)
    sender = box.address(msg_id)
    box.delete(msg_id)
    for word in nasty_words:
        msg = msg.replace(word, "XXX")
    messaging.sms_send(PHONE_NUMBER, msg)
    print "Message from %s forwarded to %s" %\
                        (sender, PHONE_NUMBER)

box = inbox.Inbox()
box.bind(message_received)
print "Gateway activated"
app_lock = e32.Ao_lock()
app_lock.wait()
```

This script should disable any notifications caused by arriving SMS messages. As in the previous example, the function `message_received()` is called when a new message arrives.

The function `message_received()` fetches the contents of the new message and deletes it immediately from the inbox. Nasty words are replaced with 'XXX' and the cleaned message is forwarded to another phone, specified in the `PHONE_NUMBER`. A log message is printed out for each forwarded SMS, which shows both the sender's and the recipient's phone numbers.

Note that you should not specify your own number in the `PHONE_NUMBER`. This would make the program forward messages to itself, resulting in an infinite loop!

4.5 SMS Game Server

We have already covered a number of new concepts in this chapter. We have learnt how to use the standard user interface framework, how to manipulate strings and how to read and receive SMS messages. In the concluding example, all these features will be put to good use.

The application is an SMS game server with a user interface. It implements the classic game of Hangman, which is described by Wikipedia as follows:

> One player thinks of a word and the other tries to guess it by suggesting letters. The word to guess is represented by a row of dashes, giving the number of letters. If the guessing player suggests a letter which occurs in the word, the other player writes it in all its correct positions. The game is over when the guessing player completes the word or guesses the whole word correctly.

Note that our implementation omits the drawing part of the game. Adding graphics to the game would be an interesting exercise after you have read Chapter 5, which introduces the required functions.

You are the game master who runs the server. You can input the word to be guessed by way of the user interface. Other players send text messages to your phone, either to guess individual characters (e.g. 'GUESS a'), or to guess the whole word (e.g. 'WORD cat'). After each guess, a reply message containing the current status of the word is sent back to the guesser. There is no limit to the number of players.

This application provides a fully working example of how to build SMS services of your own. You can easily adapt it to your own ideas. The concept is simple. The user interface provides an option to initialize a new game and an option to see the status of the current game. An inbox callback handles all incoming messages, parses them, modifies the game state and sends a reply to the guessing player.

The implementation contains some features which have not been introduced earlier, but they are presented in the following discussion. Since the code listing does not fit into one page nicely, we have divided it into three parts. You should combine the parts to form the full application.

Example 19: Hangman server (1/3)

```
import inbox, messaging, appuifw, e32, contacts

current_word = None
guessed = None
num_guesses = 0
def new_game():
    global current_word, guessed, num_guesses
    word = appuifw.query(u"Word to guess", "text")
    if word:
        current_word = word.lower()
        guessed = list("_" * len(current_word))
        num_guesses = 0
        print "New game started. Waiting for messages..."

def game_status():
    if current_word:
```

```
        appuifw.note(u"Word to guess: %s\n" % current_word +\
                     "Current guess: %s\n" % "".join(guessed) +\
                     "Number of gusses: %d" % num_guesses)
    else:
        appuifw.note(u"Game has not been started")

def quit():
    print "HANGMAN SERVER EXITS"
    app_lock.signal()
```

The first section of the code contains the functions to handle UI events: `new_game()`, `game_status()` and `exit()`. The state of the game is maintained in three variables:

- the string `current_word`, which contains the word to be guessed
- the list `guessed`, which contains the already guessed characters in their correct positions
- the integer `num_guesses`, which contains the current number of guesses in total.

It is convenient to handle `guessed` as a list instead of a string, since we cannot modify individual characters of a string without making a new copy of it. In contrast, list items can be modified without any restrictions.

Notice how the `guessed` list is created in function `new_game()`: you can use the multiplication operator to make a string of a repeating character or substring. Once we have created a string containing as many dashes as there are characters in `current_word`, we can convert it to a list of characters with function `list()`. For printing, the list is later converted back to a string with the `join()` function.

The three game state variables are shared among all functions in the game source code. In contrast, some variables, such as `word` in the function `new_game()`, are used locally in one function. Python makes a distinction between global and local variables: each variable assignment is assumed to be local to a function unless it is explicitly declared as global. This distinction is explained in the language lesson.

Python Language Lesson: local and global variables

By default, variables that are introduced in functions have only a limited lifetime and visibility. They are visible only in the function in which they were created and they disappear when execution returns from the function (see Figure 4.5).

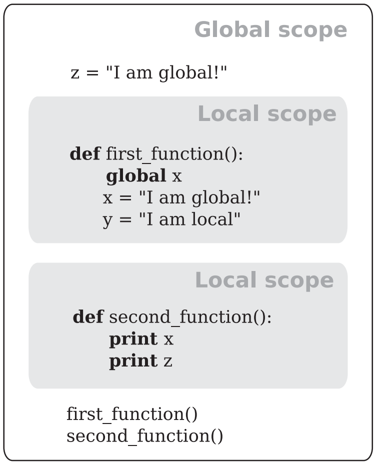


Figure 4.5 Scope diagram

In Figure 4.5, the variable `y` is local to the function `first_function()`. You cannot use it outside this function. In contrast, keyword `global` declares that the variable `x` belongs to the global scope, which covers all functions within a module. For this reason, you can use `x` also in the function `second_function()`. If a variable is defined outside any function, it belongs automatically to the global scope, such as the variable `z`.

A good rule of thumb is that if the same variable must be used in more than one function, it must be declared as global or it must be passed as a parameter to the functions.

Note that you have to assign a value to a variable before it can be used. It is meaningless to refer to a variable which does not have any content. Thus, in the example above you must call the function `first_function()` before calling `second_function()` or Python will complain `NameError: global variable "x"is not defined`. This error means that no value has been assigned to the variable yet or it was not declared as global.

Example 20: Hangman server (2/3)

```
def find_number(sender):
    cdb = contacts.open()
    matches = cdb.find(sender)
    if matches:
        num = matches[0].find("mobile_number")
        if num:
            return num[0].value
        else:
```

```
            return None
        return sender

def message_received(msg_id):
    global guessed, num_guesses
    box = inbox.Inbox()
    msg = box.content(msg_id).lower()
    sender = box.address(msg_id)
    box.delete(msg_id)

    print "Message from %s: %s" % (sender, msg)

    if current_word == None:
        return

    elif msg.startswith("guess") and len(msg) >= 7:
        guess = msg[6]
        for i in range(len(current_word)):
            if current_word[i] == guess:
                guessed[i] = guess
        num_guesses += 1

    elif msg.startswith("word"):
        if  msg[5:] == current_word:
            appuifw.note(u"%s guessed the word!" % sender)
            guessed = list(current_word)

    num = find_number(sender)
    if num:
        messaging.sms_send(num, u"Status after %d guesses: %s" %\
                                 (num_guesses, "".join(guessed)))
```

The second section of the code contains the game logic embedded in the callback function `message_received()`, which handles incoming SMS messages. In this function, we first fetch the message contents to the string `msg` and convert it to lower case. If the message arrives before a new game has been initialized, the variable `current_word` is still in its initial value `None` and the function returns immediately.

We recognize two types of message: the first follows format 'GUESS x' where 'x' is any single character. The second format is 'WORD someword' where 'someword' is the player's guess for the full word. The function `startswith()` is used to distinguish the two cases.

In the first case, we make sure that the message actually contains some character after the word 'GUESS': this is true only if the message length is at least seven characters. We reveal the guessed characters in the list `guessed` by checking in which positions the guessed character `guess` occurs in the original word `current_word`. We use a simple `for` loop to go through the individual characters. Nothing is changed if the character does not occur in the word at all.

In the second case, the player tries to guess the full word, so we check whether the word in the message `msg[5:]` (we omit the substring `"WORD"`) matches the original word `current_word`. Note that here we

do not need additional checks for the message length, since expression msg[5:] returns an empty string if the length is fewer than six characters.

At the end of the function message_received(), we send a reply message to inform the guesser about the current status of the game. However, there is a small catch: the Inbox object's function address(), which should return information about the sender of the message, does not always return the sender's phone number. If the sender exists in the address book of your phone, the name is returned instead of the phone number.

This might be convenient for an application that shows the sender information to the user but, in this case, it makes replying impossible if the sender's number exists in the address book. Luckily, there is a workaround: we use the contacts module of PyS60 to access the phone's address book, from which the sender's actual phone number can be found. This operation is performed in the function find_number. We do not explain it in detail here, as the contacts module is introduced in Chapter 7.

Example 21: Hangman server (3/3)

```
box = inbox.Inbox()
box.bind(message_received)

appuifw.app.exit_key_handler = quit
appuifw.app.title = u"Hangman Server"
appuifw.app.menu = [(u"New Game", new_game),
                    (u"Game Status", game_status)]

print "HANGMAN SERVER STARTED"
print "Select 'Options -> New Game' to initialize a new game"

app_lock = e32.Ao_lock()
app_lock.wait()
```

The final section of the game source code is simple: it just constructs the application user interface using the techniques introduced earlier in this chapter. Here we also bind the message_received() function to the Inbox, so the incoming messages are captured correctly.

Now, let us assume that you initialize a new game and choose 'code' as the hidden word. The game proceeds as follows – here the G lines denote replies by the game server and P the player's messages.

```
P: GUESS i
G: Status after 1 guesses: _ _ _ _
P: GUESS o
G: Status after 2 guesses: _ o _ _
P: GUESS r
G: Status after 3 guesses: _ o _ _
P: GUESS e
```

```
G: Status after 4 guesses: _ o _ e
P: GUESS c
G: Status after 5 guesses: c o _ e
P: WORD code
G: Status after 6 guesses: c o d e
```

The correct guess in the end triggers a dialog that shows the winning player's phone number. After this, you can either start a new game or close the application. Now start spreading the word about your revolutionary new SMS game!

4.6 Summary

In this chapter we showed how to build real S60 applications that include titles, menus and dialogs. Without this skeleton, building highly functional applications would be cumbersome. The application framework came in useful immediately when we started to search and sort your SMS inbox.

With the techniques in this chapter, especially in the concluding example, you can start building various kinds of SMS services. If you want some further exercises, see the Wikipedia article about word games and use the Hangman example as a basis for implementing other games.

In this chapter, you also learned a great deal about the Python language, including functions, callback functions, objects, tuples and the scope of variables – in other words, almost everything needed in this book. In Chapter 6, two more lessons are presented. If you feel exhausted by these lessons, we can promise that Chapter 5 will not increase your burden.

5

Sound, Interactive Graphics and Camera

In this chapter you learn how to program your phone to record and play sounds and to use its built-in text-to-speech engine. You'll see how to use the camera to take photos, either manually or fully automatically. The photos can be manipulated in various ways, for example by scaling, rotating or imprinting text on them with the image-processing features of PyS60.

We spice up the chapter with some interactivity: we show how to access the keyboard keys in three different ways. To enable you to build visually appealing applications, we show how to draw text and basic graphics, such as squares and circles, to the screen, and use them as building blocks for more complex graphics. We also make some reference to 3D graphics and, at the end of the chapter, we build an action-packed mobile game.

This is useful knowledge for creating your own music players, camera applications, games and software that can visualize data in real-time. But even more, you may get some idea of how you could turn your mobile phone into an artistic tool.

We hope that this chapter will unleash your creativity!

5.1 Sound

5.1.1 Text to Speech

Let's start with a smooth introduction to sound. Most of the latest S60 phones have text-to-speech functionality built-in, meaning that the phone can speak aloud any given string. Hopefully you can come up with some useful and amusing application for this feature.

The module named `audio` provides access to the text-to-speech engine. The function `audio.say()` takes a string as its parameter and speaks it aloud (see Example 22).

Example 22: Text to speech

```
import appuifw, audio

text = appuifw.query(u"Type a word:", "text")
audio.say(text)
```

The word you type in the input dialog is spoken aloud by the phone.

5.1.2 Playing MP3 Files

The S60 platform supports a wide range of sound formats, such as MP3, WAV, AMR, AAC, MIDI and Real Audio. Example 23 shows how to play a MP3 file.

Example 23: MP3 player

```
import audio

sound = audio.Sound.open("E:\\Sounds\\mysound.mp3")

def playMP3():
    sound.play()
    print "PlayMP3 returns.."

playMP3()
```

You must have an MP3 file named `mysound.mp3` on your phone, on the memory card in this example. If the file is missing, the example raises an exception. In Chapter 6, we explain the directory hierarchy in detail, so you know where to load and save files – here we just use a standard location for sounds.

The call to the function `audio.Sound.open()` opens the specified file, in this case `E:\Sounds\mysound.mp3` and creates an `audio.Sound` object (which we have called `sound`). When we call the function `sound.play()`, the MP3 file starts playing and plays from start to finish. In Section 5.1.5, we see how to stop the sound at any time when it is being played.

Example 23 prints out `"PlayMP3 returns.."` when the sound starts playing, as the `play()` function does not wait until the sound has reached the end. This is desirable if you want to leave the sound playing in the background and continue doing other things.

In some cases, however, you would like the `play()` function to block the program until the sound is played till the end. For example, this

behavior might be useful in a game where short sound effects are being played. Example 24 shows how to do that.

Example 24: Blocking MP3 player

```
import audio, e32

snd_lock = e32.Ao_lock()

def sound_callback(prev_state, current_state, err):
    if current_state == audio.EOpen:
        snd_lock.signal()

def playMP3():
    sound = audio.Sound.open("E:\\Sounds\\mysound.mp3")
    sound.play(callback = sound_callback)
    snd_lock.wait()
    sound.close()
    print "PlayMP3 returns.."

playMP3()
```

Here we use the `e32.Ao_lock()` object to wait for an event to occur, in a similar way to the applications in Chapter 4. The `play()` function may be given a callback function as a parameter, which is named `sound_callback()` here. The callback function is called whenever the sound state changes. The function is given three parameters: `prev_state`, `current_state` and `err`. We can check the current state of the sound with the variable `current_state`. It equals `audio.EOpen`, if the sound has finished, or `audio.EPlaying` if the sound is playing.

In the above example, we want the `play()` function to wait until the sound has finished. We use the `e32.Ao_lock` object to wait until the callback function receives a notification that the sound has finished, after which it releases the lock using the `signal()` function. As a result, `"PlayMP3 returns.."` is printed out only after the sound has been played in full.

You could easily extend the previous examples into a fully fledged MP3 player. For example, you can add a popup dialog that lets you pause, stop and fast forward the playing song. The rest of this section introduces some useful functions to do this. The next step might be to add simple playlist functionality to the player. For this purpose, you might want to have a look at Chapter 6, which introduces data handling.

5.1.3 Playing MIDI Files

Even though MP3 files are abundant nowadays, the MP3 format is not suitable for every purpose. To demonstrate how to play other sound

formats besides MP3, Example 25 uses a Musical Instrument Digital Interface (MIDI) file.

As the name suggests, a MIDI file contains commands that musical instruments transmit to control various attributes of music, such as playing notes and adjusting an instrument's sound in various ways.

Playing MIDI files works in exactly the same way as playing MP3 files. As in the previous example, your sound file should be placed on your memory card.

Example 25: MIDI player

```
import audio

midi = audio.Sound.open("E:\\Sounds\\mysound.mid")

def playMIDI():
    midi.play()
    print "PlayMIDI returns.."

playMIDI()
```

As you can see, the `audio` module takes care of handling different types of sounds automatically, so you can open any sound file that is supported by your phone in a similar way. A blocking MIDI player would look exactly like the blocking MP3 player in Example 24.

5.1.4 Additional Information on Playing Sounds

Besides the `callback` parameter that was demonstrated in Example 24, you can specify how many times the sound should be played. For instance, if you specify `play(times = 20)`, the sound will play 20 times. If you want to repeat the same sound forever, use `play(audio.KMdaRepeatForever)`. If you want to leave some silence between subsequent plays, use the `interval` parameter to specify the interval in microseconds.

Some phones do not support simultaneous playing of audio files. If you try to call the `play()` function in such a phone while a sound is playing, an exception is raised. You can prevent this from happening by always calling `stop()` before playing a sound. Also note that you have to call `stop()` explicitly before your script exits or the sound will keep playing in the background even after your program has ended.

5.1.5 Recording Sounds

Now we turn your phone into a lo-fi music studio by showing how to record sounds with the `audio` module. Although a phone supports playing a wide range of different sound formats, it can typically record only WAV and AMR files.

Sounds are recorded using the same `audio.Sound` object that was used for playback. In Example 26, we make a simple program that lets you record and play a single sound.

Example 26: Sound recorder

```
import appuifw, audio, os

MENU = [u"Play sound", u"Record sound", u"Delete sound"]
SOUNDFILE = u"E:\\sound.wav"
sound = None

while True:
    index = appuifw.popup_menu(MENU, u"Select operation")
    if sound:
        sound.stop()
    sound = audio.Sound.open(SOUNDFILE)
    if index == 0:
        sound.play()
    elif index == 1:
        sound.record()
        appuifw.query(u"Press OK to stop recording", "query")
        sound.stop()
    elif index == 2:
        os.remove(SOUNDFILE)
    else:
        break
```

The program starts by displaying a popup menu, as shown in Figure 5.1, that lets the user choose the desired operation. If you attempt to play a sound before one has been recorded, the program crashes as the file `SOUNDFILE` cannot be found. Chapter 6 gives you some ideas on how to handle situations like this properly.

The function `sound.record()` starts recording. The recording continues until `sound.stop()` is called. Since we want the user to decide when to stop, we present a dialog that blocks the execution until the user selects OK.

If the sound file already exists when recording starts, the new sound is appended to the end of the existing file. You can hear this easily by repeating the 'Record sound' operation several times and then listening to the sound. If you want to start recording from scratch, the previous file must be deleted first. This can be done with the `os.remove()` function, which deletes the file whose name is given in the parameter. This function is called if the user chooses the 'Delete sound' operation from the menu. Note that you need to import the `os` module to use this function.

As the program operates within an infinite loop, the popup menu is shown again once an operation finishes. Note that `sound.play()` only starts the playing and does not wait for it to finish, so a new dialog may be shown although the sound is still playing. We stop playing, however, once the next operation has been chosen, as it is not a good idea to



Figure 5.1 Sound recorder

execute several `record()` and `play()` operations simultaneously on the same sound object. The program exits when you cancel the popup menu.

Recording Animal Sounds

Example 27 extends the previous sound recorder by recording sounds to several sound files instead of only one. The example lets you record and play sounds of animals, namely the sound of a dog, a cat and a cow. If you have a toddler, this might be a good way to introduce the wonderful world of mobile programming.

Example 27: Animal sounds

```
import appuifw, audio

animals = [u"dog", u"cat", u"cow"]

def record_animal_sounds():
    for animal in animals:
        noise = audio.Sound.open("e:\\" + animal + ".wav")
        if appuifw.query(u"Record sound of a " + animal, "query"):
            noise.record()
            appuifw.query(u"Press OK to stop recording", "query")
            noise.stop()
            noise.close()

def select_sound():
    global funny_noise
```

```
    funny_noise = None
    while True:
        index = appuifw.popup_menu(animals, u"Select sound:")
        if funny_noise:
            funny_noise.stop()
        if index == None:
            break
        else:
            play_animal_sound(u'e:\\ + animals[index] + '.wav')

def play_animal_sound(soundfile):
    global funny_noise
    funny_noise = audio.Sound.open(soundfile)
    funny_noise.play()

record_animal_sounds()
select_sound()
```

This time, you have to record sounds of the three animals when the script starts. This is handled by the function `record_animal_sounds()` which loops over the list of animals, `animals`. After this, the function `select_sound()` presents a menu that lets you choose a sound to play.

Note that the sounds are saved to the root of the memory card, that is, the E: drive. If you do not have a memory card, you can change the path, for instance, to `C:\\Python`.

The playing of a sound is handled by the function `play_animal_sound()`. Note that here we declare the variable `funny_noise` as global. This variable contains the sound object that is now playing. If `funny_noise` was not global, we would not be able access it after the function `play_animal_sound()` returns, so stopping the sound would be impossible. As the variable is declared global, it stays alive even after the function returns, so we can stop the sound once the user chooses the next sound to be played. This example exits when you cancel the playback dialog.

Recording a Phone Call

Interesting things happen when you record or play a sound during a phone call. Calling `play()` plays the sound to the speech channel, meaning that those participating in the phone call can hear it. Calling `record()` starts recording the telephone call.

See Chapter 7 for more information about using the telephone.

5.1.6 Other Useful Functions of the Audio Module

A range of additional functions are provided by the audio module:

- `state()` returns the current state of the `Sound` object. The different states are returned as constants:
  - `ENotReady` – the `Sound` object has been constructed but no audio file is open.
  - `EOpen` – an audio file is open but no playing or recording operation is in progress.
  - `EPlaying` – an audio file is playing.
  - `ERecording` – an audio file is being recorded.
- `max_volume()` returns the maximum volume of the device.
- `set_volume(volume)` sets the volume of the device. If the given volume is negative, the volume is set to zero which mutes the device. If the volume is greater than the maximum volume, the maximum volume is used.
- `current_volume()` returns the current volume level.
- `duration()` returns the duration of the file in microseconds.
- `set_position(microseconds)` sets the position of the playhead.
- `current_position()` returns the current playhead position in microseconds.

5.2 Keyboard Keys

In this section, we introduce you to the keys of the phone keyboard. Although there is nothing particularly difficult in registering the click of a key, there are several things worth noticing.

First, a single physical key may produce different results based on the keyboard mode: for instance, a key might produce an upper or lower case 'a' depending whether some other key is active. Second, there are several different key events: the user can select a key, hold it down and release it – all these events can be registered separately.

With the help of the next three examples, we show three different approaches to programming keyboard keys. Each example takes a slightly different approach, but basically they all do the same thing:

- If the user presses the up-arrow key (navigation key up), a note dialog tells us that the up-arrow key was pressed, as shown in Figure 5.2.
- If the user presses keyboard key 2, a note dialog tells us that key 2 was pressed.



(a) (b)

Figure 5.2 Up arrow is pressed

Even though all three approaches are used here to accomplish the same task, each of them is best suited to handle a particular use case:

- The first approach (Section 5.2.1) is handy when you need to handle the clicks of only a few specific keys.
- The second approach (Section 5.2.2) is suitable if you need more complicated processing of several keys.
- The third approach (Section 5.2.3) is a generalization of the second and allows you to detect whether a key is clicked or continuously held down.

Before we present the approaches, note one important prerequisite: when you want to register any key event (except the left and right softkeys), you need to assign the `appuifw.Canvas` object to the application body, `appuifw.app.body`.

`Canvas` is a UI element that provides a drawable area on the screen and provides support for handling keyboard key events. The `Canvas` object can be instantiated as follows:

```
canvas = appuifw.Canvas()
appuifw.app.body = canvas
```

This creates a `Canvas` object named `canvas` which is then assigned to the application body. As you may remember, Figure 4.1 illustrates

the application body with respect to the other parts of the standard S60 application UI.

The `Canvas()` function, which creates the corresponding object, takes two optional parameters: `redraw_callback` and `event_callback`. We can assign a callback function to both of these parameters. The callback functions are called when the screen must be redrawn or when a keyboard event occurs. The `redraw_callback()` function is described in Section 5.3 and the `event_callback` function in Sections 5.2.2 and 5.2.3.

Now, let's dive into the three approaches on programming keyboard keys.

5.2.1 Binding a Keycode to a Callback Function

In this approach, we bind a callback function to respond to a specific key event. The key is identified by a *keycode*. For each key on the keyboard there exists at least one keycode that corresponds to a particular character or action, as shown in Figure 5.3. Note that a single physical key may produce many different characters; thus, many different keycodes may correspond to a single key.

The keycodes are defined as constants in the module `key_codes`. You can find an overview of often used keycodes in Figure 5.3. To use the keycodes, you need to import the `key_codes` module at the beginning of the script.

Example 28: Binding a keycode to a callback function

```
import appuifw, e32, key_codes

def up():
  appuifw.note(u"Up arrow was pressed")

def two():
  appuifw.note(u"Key 2 was pressed")

def quit():
  app_lock.signal()

canvas = appuifw.Canvas()
appuifw.app.body = canvas

canvas.bind(key_codes.EKeyUpArrow, up)
canvas.bind(key_codes.EKey2, two)

appuifw.app.exit_key_handler = quit
app_lock = e32.Ao_lock()
app_lock.wait()
```



	Keycodes	Scancodes
1	EKeyLeftSoftkey	EScancodeLeftSoftkey
2	EKeyYes	EScancodeMenu
3	EKeyStar	EScancodeMenu
4	EKey5	EScancode5
5	EKeyStar	EScancodeStar
6	EKeyLeftArrow	EScancodeLeftArrow
7	EKeyUpArrow	EScancodeUpArrow
8	EKeySelect	EScancodeSelect
9	EKeyRightArrow	EScancodeRightArrow
10	EkeyDownArrow	EScancodeDownArrow
11	EKeyRightSoftkey	EScancodeRightSoftkey
12	EKeyNo	EScancodeNo
13	EKeyBackspace	EScancodeBackspace
14	EKeyEdit	EScancodeEdit
15	EKeyHash	EScancodeHash

Figure 5.3 Overview of keycodes and scancodes

The `canvas.bind()` function binds a keycode to a callback function. For example, `canvas.bind(key_codes.EKeyUpArrow, up)` binds the callback function `up()` to the up-arrow key. This informs the canvas object that whenever the key corresponding to the keycode `EKeyUpArrow` (navigation key up) is pressed, the callback function `up()` should be called. Similarly, we bind the keycode `EKey2` to the function `two()`.

In some cases, you may want to cancel the binding. This can be accomplished by binding the value `None` to a keycode, in place of a callback function.

5.2.2 Using the `event_callback()` Function

In the second approach, we use a single callback function named `keys()` to handle all key events. Depending on the parameter `event` that is given to the function, we take a corresponding action. We use the `Canvas` object's `event_callback` parameter to direct all events to a single function, `keys()`.

Example 29: Using the `event_callback()` function

```
import appuifw, key_codes, e32

def keys(event):
    if event['keycode'] == key_codes.EKeyUpArrow:
        appuifw.note(u"Up arrow was pressed")
    elif event['keycode'] == key_codes.EKey2:
        appuifw.note(u"Key 2 was pressed")

def quit():
  app_lock.signal()

canvas = appuifw.Canvas(event_callback = keys)
appuifw.app.body = canvas
appuifw.app.exit_key_handler = quit
app_lock = e32.Ao_lock()
app_lock.wait()
```

When the user presses a key, the canvas receives a key event and generates a dictionary object which is handed to the `keys()` function through the parameter `event`. This object holds information about the key that produced the event and whether the key was pressed or released.

In essence, a dictionary object is a collection of key–value pairs. This data structure might be familiar to you from other programming languages, such as Java's Hashtable or PHP's or Perl's hash arrays. The dictionary object is introduced more thoroughly in Chapter 6 – here

we only peek inside the contents of one particular dictionary object, `event`.

The event callback function receives a single parameter that is a dictionary object, `event`. It includes the following information:

- `keycode` is an identifier for the keyboard key event. Some physical keys on the keyboard may correspond to several keycodes, for instance, to upper-case and lower-case A. Thus, if your application must recognize a specific character, keycodes are a suitable choice. Keycode names are defined in the `key_codes` module and they start with the letters `EKey`.
- `scancode` is a lower-level identifier of the physical keyboard key. Scancode names start with the letters `EScancode`. If your application must recognize events related to a physical key, scancodes are a good choice. See Figure 5.3 to see some cases where the keycode and scancode do not map to the same key. You can use scancodes wherever keycodes are used, for instance, you could have used them in Example 28.
- `modifiers` returns modifier keys that apply to this key event, such as Ctrl or Alt. Modifiers are seldom used on a mobile phone keyboard.
- `type` indicates whether the key has just been pressed down (`appuifw.EEventKeyDown`), is down (`appuifw.EEventKey`), or has been released (`appuifw.EventKeyUp`). We can use this information to distinguish between single clicks and continuous key presses (see Example 30).

You can fetch the above values from the dictionary object as follows:

```
ev_keycode = event["keycode"]
ev_scancode = event["scancode"]
ev_modifiers = event["modifiers"]
ev_type = event["type"]
```

To detect which key is being pressed, we simply compare the keycode or scancode value from the `event` dictionary with the code that we want to detect. When the right softkey is pressed, the `appuifw.app.exit_key_handler()` callback is always executed.

5.2.3 Key Pressed or Held Down

The third approach shows how to distinguish whether a certain key was clicked once or whether it is being held down. This can be useful, for

instance, in a game where a battleship is steered with arrow keys. In this case, being unable to see the distinction between a click that corresponds to a small adjustment and a continuous move to the right is likely to cause a shipwreck.

Example 30: Key pressed or held down

```
import appuifw, e32, key_codes

key_down = None
clicked = None

def handle_event(event):
    global clicked, key_down
    if event["type"] == appuifw.EEventKey:
        if key_down:
            key_down = (event["keycode"], "down")
        else:
            key_down = (event["keycode"], "pressed")
    elif event["type"] == appuifw.EEventKeyUp and key_down:
        code, mode = key_down
        if mode == "pressed":
            clicked = code
        key_down = None

def key_clicked(code):
    global clicked
    if code == clicked:
        clicked = None
        return True
    return False

def key_is_down(code):
    if key_down and key_down == (code, "down"):
        return True
    return False

def quit():
    global running
    running = False

canvas = appuifw.Canvas(event_callback = handle_event)
appuifw.app.body = canvas
appuifw.app.exit_key_handler = quit

running = True
while running:
    e32.ao_sleep(0.1)
    if key_clicked(key_codes.EKeyUpArrow):
        appuifw.note(u"Up arrow was pressed")
    elif key_is_down(key_codes.EKey2):
        canvas.clear((0, 0, 255))
    else:
        canvas.clear((255, 255, 255))
```

This example is somewhat more complex than the previous ones. Do not worry if you cannot understand this script instantly. You can always

come back to it later once you have advanced in the book and have learned more about Python.

This example uses an event callback function that is similar to the second approach. In contrast to the second approach, however, here we use the event type in `event["type"]` to distinguish between the states of a key.

These event-handling functions are not application-specific, but are usable in any application. This is in contrast to both the previous approaches, which included application-specific actions (showing the dialogs) in the event-handling function. Because of this, you can use the functions `handle_event()`, `key_clicked()` and `key_is_down()` in your own applications.

The event callback function `handle_event()` works as follows: if the key is down (event type `appuifw.EEventKey`), we make a distinction between the first key-down event, when the variable `key_down` equals `None` and further events that happen only if the key is not released immediately. We mark the first event with `"pressed"` and the following key-down events with `"down"`. When the key is released (event type `appuifw.EEventKeyUp`) we check the mode: if the key was `"pressed"`, the user has only clicked the key and we save the keycode to the variable `clicked`. If the mode was `"down"`, this event means that the key was just released and we reset the `key_down` variable back to `None`.

Given a keycode, the function `key_clicked()` checks whether the last clicked key in the variable `clicked` equals the parameter and then resets `clicked`. Thus, with this function you can check whether a certain key was clicked.

In a similar manner, we check whether the key that is now held down (if any) equals the parameter in the function `key_is_down()`. The function returns `True` as long as the particular key is held down.

As with the previous approaches, a click of the up-arrow key shows a dialog. If you hold the key down, instead of just clicking it, no dialog is shown. On the other hand, if you hold the 2 key down, the screen turns blue. However, if you just click the 2 key, nothing happens. We could have shown another dialog when the 2 key is held down, but seeing many dialogs in a row saying that the key is still down would be rather annoying. That is why the screen is turned blue instead.

Later on, we use this approach in a drawing program (Example 33).

5.2.4 Capturing any Key Event on Your Phone

The `keycapture` module offers an API for global capturing of key events. With this module, you can react to key events even if some other S60 application is active and your PyS60 program is running only in the background. For instance, you could have a key combination that triggers an action regardless of the application that you are using on the phone.

The `keycapture` module provides a `KeyCapturer` object that is used for listening to the events by way of a callback function. The callback is called each time any of the specified keys is pressed.

Since capturing all key presses on your phone has security and privacy implications, 3rd Edition phones require a special capability (`SwEvent`) to use this module. See Appendix A for more information about capabilities.

5.3 Graphics

When we want to display 2D graphics or images on the screen, the `Canvas` object is needed in the application body. `Canvas` is a UI element that provides a drawable area on the screen but it also provides support for handling keyboard events, as we saw in Section 5.2. We showed how to create a canvas object and how to assign it to the application body. We also mentioned that it has an optional parameter, `redraw_callback`, that defines a callback function that is called whenever a part of the canvas must be redrawn. Typically, this happens when the user switches away from the Python application and back again or after a popup menu is displayed.

5.3.1 Drawing Graphics Primitives

This section shows you how to draw circles, rectangles, lines and points – that is, all kinds of graphics primitives. A common way to perform drawing and showing graphics on the screen is that you first create a `graphics.Image` object, manipulate that object and then draw it on the canvas (screen) in the `redraw_callback` function.

This process is called *double buffering*. The name refers to the fact that instead of drawing primitives directly on the canvas, which is possible as well, you draw them in a separate image (buffer) first. This way you do not have to draw everything again when the canvas must be redrawn, for example after a popup dialog has cleared a part of the canvas. Instead, you simply copy the image to the canvas. The `Image.blit()` function that handles the copying is typically accelerated by hardware and is, thus, fast.

To do this, we need to import the `graphics` module, which gives access to functions for drawing graphics primitives and loading, saving, resizing and transforming images. It is loosely based on the Python Imaging Library (PIL), though it supports only a restricted set of its functions.

Let's have a look at the `graphics` module. Example 31 draws a red point, a yellow rectangle and some white text to the screen, as shown in Figure 5.4. We use keyboard keys to draw one of these graphics primitives on the screen or to draw all primitives at the same time.

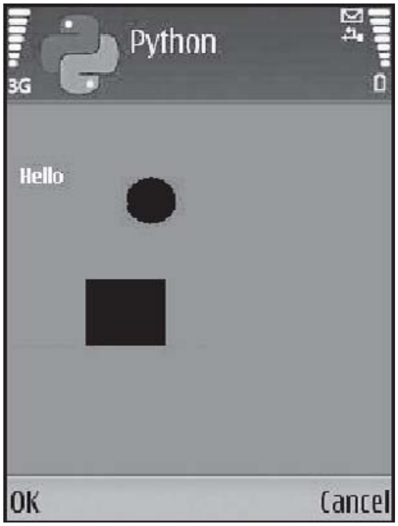


Figure 5.4 Graphics primitives drawn to the screen

Example 31: Graphics primitives

```
import appuifw, e32, key_codes, graphics

WHITE = (255,255,255)
RED = (255,0,0)
BLUE = (0,0,255)
YELLOW = (255,255,0)

def draw_rectangle():
    img.rectangle((50,100,100,150), fill = YELLOW)

def draw_point():
    img.point((90,50), outline = RED, width = 30)

def draw_text():
    img.text((10,40), u"Hello", fill = WHITE)

def handle_redraw(rect):
    if img:
        canvas.blit(img)

def handle_event(event):
    ev = event["keycode"]
    if event["type"] == appuifw.EEventKeyDown:
        img.clear(BLUE)
    if ev == key_codes.EKeyUpArrow:
        draw_point()
    elif ev == key_codes.EKeyRightArrow:
        draw_text()
    elif ev == key_codes.EKeyDownArrow:
        draw_rectangle()
    elif ev == key_codes.EKeyLeftArrow:
```

```
        draw_point()
        draw_text()
        draw_rectangle()
    handle_redraw(None)

def quit():
    app_lock.signal()

img = None
canvas = appuifw.Canvas(redraw_callback = handle_redraw,\
                         event_callback = handle_event)
appuifw.app.body = canvas
appuifw.app.screen = "full"
appuifw.app.exit_key_handler = quit

w, h = canvas.size
img = graphics.Image.new((w, h))
img.clear(BLUE)

app_lock = e32.Ao_lock()
app_lock.wait()
```

At the beginning of the script we assign various colors to constants: WHITE, RED, BLUE and YELLOW. The color representation consists of a three-element tuple of integers in the range from 0 to 255, representing the red, green and blue (RGB) components of the color.

If you have written some HTML code, you may be familiar with the hexadecimal representation of a color: in this case, the color is specified as a value such as `0xRRGGBB`, where `RR` is the red, `GG` the green and `BB` the blue component of the color, each of which is a hexadecimal value from 0x0 to 0xff (0–255). Using this notation, we could specify, for instance, `YELLOW = 0xffff00`.

Next we define three functions, each of which handles the drawing of a single primitive:

- `draw_rectangle()`: the first tuple specifies the top-left and lower-right corners of the rectangle, in format (x1, y1, x2, y2) and the `fill` parameter defines its color.
- `draw_point()`: the first tuple specifies the center of the point at (x,y). The parameter `width` specifies the size in pixels and `outline` specifies the color.
- `draw_text()`: draws the Unicode string `u"Hello"` on image at the specific location that is defined in the first tuple.

Toward the bottom of the script you can see the following lines:

```
w, h = canvas.size
img = graphics.Image.new((w, h))
```

They create a new `Image` object whose size equals to the canvas. Before these lines, notice that we create a `Canvas` object and assign two callback functions to it, which handle redrawing of the canvas and key events, correspondingly. As you might guess, the function `img.clear(BLUE)` clears the image to blue initially. Note that the image should be created only after the `Canvas` object is assigned to the application body. Otherwise `canvas.size` may return an incorrect tuple for the screen size.

The function `handle_event()` draws the primitives on the image, `img`, depending on which key is pressed. When any key is pressed down, the image is first cleared for drawing. Then, a point, text, a rectangle or all of these are drawn in the respective functions, based on the actual keycode. Finally, we request the canvas to be redrawn by calling the function `handle_redraw()`.

The function `handle_redraw()` is simple: it gets a single parameter, `rect`, that defines the area on the screen that must be redrawn. For simplicity, we may omit this parameter and always redraw the whole screen. A performance-critical application, say a game, might use the `rect` parameter to speed up redrawing if only a fraction of the canvas must be refreshed.

Both the `Image` and `Canvas` objects have a `blit()` function that is used to copy one image to another. In this case, we copy `img` to the canvas as whole, thus no additional parameters are specified for the function. The `if` clause ensures that no blitting is performed until `img` has been initialized appropriately.

5.3.2 More on Graphics Primitives

Besides the primitives `point`, `rectangle` and `text`, there are other primitives available, such as `line`, `polygon`, `ellipse` and `pieslice`. You can also change the font and size of the `text` primitive. For more information, see the PyS60 documentation. The `graphics` module also offers several image-manipulation methods for resizing, flipping and rotating images, which are described in the documentation.

Chapter 11 contains more advanced examples that use the `graphics` module. In particular, have a look at Section 11.3, which describes an artistic tool called MobileArtBlog in detail. The tool combines the power of the camera and graphics modules in an innovative way: you can pick patterns from your physical environment and use them as paint brushes!

With the help of the graphics primitives, you can also design UI elements of your own, instead of using the native UI elements of the S60 platform. This is demonstrated in Sections 11.2 and 11.5.

5.3.3 Loading and Saving Images

Instead of using graphics primitives, you can use photos and other graphical material in the JPEG or PNG formats. The handling of these materials is done through the `Image` object in a similar way to the way graphics primitives are handled. Note that once a pre-made image has been loaded, you can use it in the same way as any other `Image` object, for example, any of the above graphic primitives can be drawn on it.

A pre-made image is loaded and saved to a new file as follows:

```
img = graphics.Image.open("e:\\Images\\picture.jpg")
img.save("e:\\Images\\picture_new.jpg")
```

5.3.4 Image Masks

When you copy one image to another using the `blit()` function, the shape of the source image is rectangular by default. In some cases, this is not desirable and you would like to mask out certain parts of the source image.

For this purpose, you need a black and white mask image with the visible parts painted in white. Figure 5.5 shows an example: on the left, there is the original image and, on the right, the corresponding mask that masks out the background of the arrow.



Figure 5.5 An image and its corresponding mask

Then, you can load and use the mask to copy only the visible parts, defined by the mask, `mask_img`, of the source image, `src_img`, to the canvas as follows:

```
mask_img = Image.new(size = (50, 50), mode = '1')
mask_img.load('e:\\Images\\mask_img.png')
src_img = graphics.Image.open('e:\\Images\\orig_img.png')
canvas.blit(src_img, target=(0,0), source=(0,0), mask = mask_img)
```

The parameter `mode = "1"` specifies that the new image, `mask_img`, has only two colors, black and white. This mode is required for masks.

Section 11.5 presents an example that uses masks extensively.

5.3.5 Taking a Screenshot

The view that is visible on the phone's screen can be captured with the function `graphics.screenshot()`. This function converts the current

view to a new `Image` object and returns it to the caller. Example 32 takes a screenshot of itself and saves it to a file.

Example 32: Screenshot

```
import graphics

print "Hi there!"
img = graphics.screenshot()
img.save(u"e:\\Images\\screenshot.png")
```

Note that if you have a new memory card, it might not have the Images directory. In this case, use the phone's default camera application to shoot and save a photo to the memory card. This will create the required directory.

Screenshots that appear in this book were taken as follows: we used the `e32.Ao_timer` object to call a function after a certain time period – see Example 50 for a similar usage of the timer. The function contained the same lines as the example above. For each figure, we calibrated the delay so that the screen would contain the exact arrangement we wanted to capture.

5.3.6 Interactive Graphics

In Example 33, we learn how to move a single black point, leaving traces, on a white background (someone might call this a pen), as shown in Figure 5.6. This example combines earlier lessons learnt, namely drawing graphics primitives and handling keyboard events.

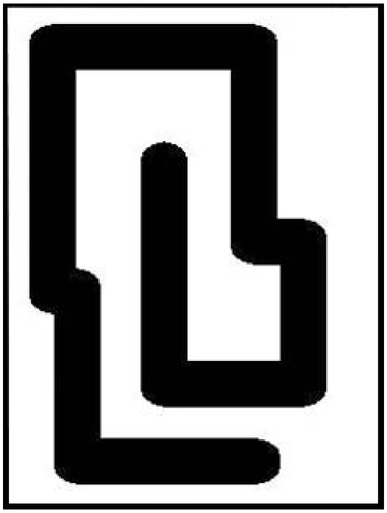


Figure 5.6 Moving dot with its trace

Example 33: Moving graphics

```
import appuifw, graphics, e32, key_codes

BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
key_down = None
clicked = None

def handle_event(event):
    global clicked, key_down
    if event['type'] == appuifw.EEventKey:
        if key_down:
            key_down = (event['keycode'], "down")
        else:
            key_down = (event['keycode'], "pressed")
    elif event['type'] == appuifw.EEventKeyUp and key_down:
        code, mode = key_down
        if mode == "pressed":
            clicked = code
        key_down = None

def key_is_down(code):
    if key_down and key_down == (code, "down"):
        return True
    return False

def quit():
    global running
    running = False

def handle_redraw(rect):
    if img:
        canvas.blit(img)

img = None
canvas = appuifw.Canvas(event_callback = handle_event,
                        redraw_callback = handle_redraw)
appuifw.app.screen = "full"
appuifw.app.body = canvas
appuifw.app.exit_key_handler = quit

x = y = 100
w, h = canvas.size
img = graphics.Image.new((w, h))
img.clear(WHITE)

running = True
while running:
    if key_is_down(key_codes.EKeyLeftArrow): x -= 5
    elif key_is_down(key_codes.EKeyRightArrow): x += 5
    elif key_is_down(key_codes.EKeyDownArrow): y += 5
    elif key_is_down(key_codes.EKeyUpArrow): y -= 5

    #img.clear(WHITE)
```

```
    img.point((x, y), outline = BLACK, width = 50)
    handle_redraw(None)
    e32.ao_yield()
```

The functions `handle_event()` and `key_is_down()` are familiar from Example 30. Again, we create a separate image, `img`, on which the dot is first drawn and then the whole image is copied onto the `canvas` in `handle_redraw()`.

The dot is moved with the arrow keys. Depending on which arrow is held down, we move the dot's location the corresponding direction in `x` and `y` coordinates. Here the dot is moved 5 pixels at time, but you can experiment with different values.

The `e32.ao_yield()` at the end of the loop makes sure that the system leaves some time to register the keyboard events, as drawing in the tight loop consumes lots of CPU power and might make the system unresponsive.

Actually, since we move the dot only when the user presses a key, we could have used the approach of Example 28 to handle the key events. In this case, we would not need the busy loop at the end, which would mean less computing and savings in precious battery time. However, this example should prepare you for Example 39, where the loop is actually necessary.

One line, `img.clear(WHITE)`, is preceded by the hash mark (`#`) which means that the line is a comment and is ignored by the PyS60 interpreter. If you remove the hash mark, the line is executed and the dot does not leave any traces. Note, however, that clearing the whole image because the dot has moved one step forward is unnecessarily expensive. A better, but less straightforward, approach for clearing traces would be to draw a small white rectangle on top of the old dot before drawing the new dot.

5.3.7 3D Graphics

Python for S60 provides support for 3D graphics by way of the modules `gles` and `glcanvas`. The `gles` module contains Python bindings to the OpenGL ES 2D and 3D graphics API. The `glcanvas` module provides an object for displaying the 3D graphics, in the same way as the `Canvas` object is used to display 2D graphics.

Making 3D graphics with OpenGL is somewhat complicated. Fortunately, several good books and free tutorials have been written about the subject. Thus, we do not go into details here. A good starting point is, for instance ***http://pyopengl.sourceforge.net***, which explains a Python version of the OpenGL graphics. The Python for S60 version follows these conventions closely.

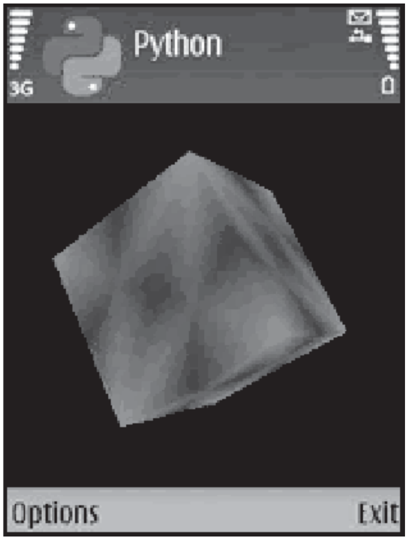


Figure 5.7 OpenGL cube

We provide a teaser for 3D graphics on this book's website. There you can find an example of a 3D cube, shown in Figure 5.7, which spins on the x, y and z axes simultaneously.

5.4 Camera

For using the built-in camera of an S60 phone, PyS60 offers a module named `camera`. We think that this is one of the most fun modules in PyS60, as it can give your programs a new view of the surrounding world.

Because of our enthusiasm for the `camera` module, this book includes seven examples that are based on taking photos. Being able to use the camera programmatically – even in the most esoteric ways – may change your relationship with camera phones in a profound manner. For instance, see Section 11.2 for a description of a large-scale urban photo hunt that was implemented using the `camera` module! The `camera` module includes three types of functions: to query features of the camera; to open and close the viewfinder; and to take photos.

Note that the camera consumes a lot of energy. If you have an application that uses the viewfinder or takes photos frequently, in many cases it is necessary to recharge the battery after four or five hours.

5.4.1 Querying Features of the Camera

The following functions are available in the `camera` module to query camera-related features of your phone:

- `cameras_available()` returns the number of cameras available in the device.
- `image_modes()` returns the image modes supported by the device as a list of strings, for example: [RGB12, RGB16, RGB] (the last one corresponds to full-color images).
- `image_sizes()` returns the image resolutions supported by the device as a list of (w, h) tuples, for example: [(640, 480), (160, 120)].
- `flash_modes()` returns the flash modes supported by the device as a list of strings.
- `max_zoom()` returns the maximum digital zoom value supported by the device as an integer.
- `exposure_modes()` returns the exposure settings supported by the device as a list of strings.
- `white_balance_modes()` returns the white-balance modes supported by the device as a list of strings.

5.4.2 Using the Viewfinder

Taking a good photo is difficult if you get no visual feedback for aiming the camera. For this purpose, the viewfinder, like the one shown in Figure 5.8, produces a stream of `Image` objects that are produced by the camera in real-time. It is up to you what to do with these images, but



Figure 5.8 Viewfinder

typically you want to show them on the canvas, to help the user aim the camera properly.

Example 34: Viewfinder

```
import camera, appuifw, e32

def viewfinder(img):
    img.point((100, 100), outline = (255, 0, 0), width = 10)
    canvas.blit(img)

def quit():
    camera.stop_finder()
    lock.signal()

appuifw.app.body = canvas = appuifw.Canvas()
appuifw.app.exit_key_handler = quit

camera.start_finder(viewfinder)
lock = e32.Ao_lock()
lock.wait()
```

The function `camera.start_viewfinder()` takes a single parameter, a callback function that handles the incoming images. In this case, the job is handled by the function `viewfinder`. To convince you that the viewfinder images are just normal `Image` objects, we draw a small red dot on each image before showing it on the canvas.

The `camera.start_viewfinder()` function accepts two optional parameters besides the callback function:

- `backlight` defines whether the device backlight is kept on when the camera viewfinder is in operation. `backlight = 1` makes it on, which is the default, and `0` off.
- `size` defines the viewfinder image size, as in `size = (176, 144)`.

It is important to stop the viewfinder, using the `camera.stop_finder()` function, once you do not need it, otherwise it may keep the camera busy in the background and drain the battery quickly.

5.4.3 Taking a Photo

The most important functionality that one expects from a camera is to take a photo. Without further ado, here is how to do it.

Example 35: Minimalist camera

```
import camera
photo = camera.take_photo()
photo.save("E:\\Images\\minicam.jpg")
```

The function `camera.take_photo()` returns the photo as an `Image` object. With `photo.save('E:\\Images\\minicam.jpg')` the photo is saved to the image gallery of the phone. If some other application is using the camera when you start this script, the program fails and an error is shown.

After trying Example 35, you are probably delighted to see Example 36, which lets you use the viewfinder before taking a photo.

Example 36: Taking photos with a viewfinder

```
import e32, camera, appuifw, key_codes

def viewfinder(img):
    canvas.blit(img)

def shoot():
    camera.stop_finder()
    photo = camera.take_photo(size = (640, 480))
    w, h = canvas.size
    canvas.blit(photo, target = (0, 0, w, 0.75 * w), scale = 1)
    image.save('e:\\Images\\photo.jpg')

def quit():
    app_lock.signal()

appuifw.app.body = canvas = appuifw.Canvas()
appuifw.app.title = u"Camera"
appuifw.app.exit_key_handler = quit

camera.start_finder(viewfinder)
canvas.bind(key_codes.EKeySelect, shoot)

app_lock = e32.Ao_lock()
app_lock.wait()
```

First we initialize the application UI and assign a canvas to the application body in a familiar way. We bind the Select key to the function `shoot()` that is used to take the actual photo – this corresponds to the first approach to handling the keyboard keys, in Section 5.2.1. The viewfinder functionality is based on Example 34.

Once the user clicks the Select key, the function `shoot()` is called. First the viewfinder is closed, which freezes the latest image on the screen, so that the user can see that the camera is taking the photo. Then we take the actual photo with the `camera.take_photo()` function. This may take a few seconds and you should hold the camera still during this time.

Once the photo, `photo`, is ready, we show it on the canvas. Note that the photo is typically taken in 3:4 picture ratio, which is not the same as the canvas size, usually. The photo is too big to fit the canvas and we do not want to resize it to fill the canvas in full, since this would break the aspect ratio. Instead, we define the target area to which the photo should be copied. The target area has the aspect ratio of 3:4, which the photo is

then resized to fit. These actions are handled by two optional parameters to the `blit()` function, namely `target` and `scale`. you can use the phone's Gallery application to see the newly taken photo or you can copy it to your PC for viewing.

You can change the exposure adjustment and the white balance settings as well as using digital zoom and flash. All these parameters are described in detail in the PyS60 API reference documentation.

5.5 Mobile Game: UFO Zapper

The concluding example of this chapter is a game. In contrast to the Hangman server that was presented in Section 4.5, this is a classic single-player action game with stunning graphics – just see Figure 5.9. Your job is to save the world from a squadron of invading UFOs with a moving pad that shoots laser beams (ahem).

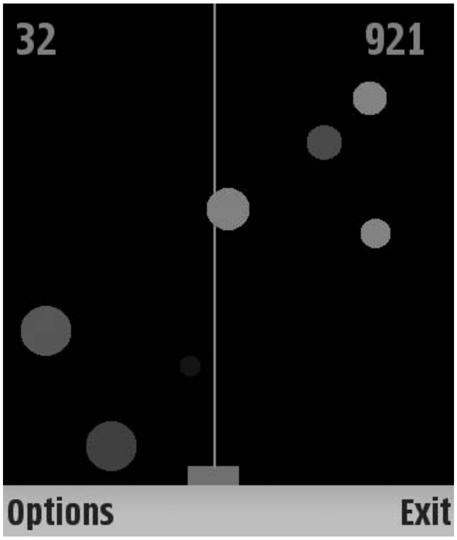


Figure 5.9 UFO Zapper

You must shoot down as many UFOs as possible within a given time limit. You get points for each hit: the smaller the UFO you hit the more points you get. Try to beat our highest score of 2304 points!

5.5.1 Structure of a Game Application

The example presents several useful concepts that are often used in game applications:

- an event loop to control the application
- dynamic time
- double buffering
- module `random`.

Using an event loop to control an application

At the beginning of Chapter 4, we noted that we have to use the `e32.Ao_lock` object to stop the execution and start waiting for the user input. However, in an action game something should be happening all the time, even if the user does nothing. Thus, instead of a lock, the game application is controlled by an *event loop*.

An event loop is typically a simple `while` loop which takes care of advancing time step by step. Like `Ao_lock`, the loop prevents the application from exiting instantly.

Whereas an animation proceeds from start to end without any user interaction, a game must react to user input. This is handled in a similar way to any other application, using event callbacks, which may change values of global variables and thus change the game state. Overall, the structure of an event loop is typically as follows:

```
Initialize event callbacks
while <end condition>:
    Update game state
    Redraw screen
    Pause
```

The last command in the event loop instructs the execution to pause for a while. We want the event loop to iterate as fast as possible, but we want to control its speed. We need to give the user some time to perceive what is happening in the game, as human perception is extremely slow compared to the 200–300 MHz CPU of a modern mobile phone. We also want to give the application some time to react to user events.

For the above reasons, we pause the event loop for a short while after each iteration. The `e32.ao_sleep()` function pauses the execution for the given time, which can be a fraction of a second. In this case, we pause for a hundredth of a second, `e32.ao_sleep(0.01)`, after each iteration. This means that the game is capable of showing 100 frames per second (FPS) at best.

Dynamic time

In some cases we want to give the user more time to observe the situation. In this game, we increase the length of the pause to a tenth of a second when a laser beam is fired, so that the user can see how many hits she

scored. The ability to change the length of the pause in the event loop is called *dynamic time*.

In an extreme case, we might want to pause for the minimum possible amount of time, 0 seconds, but let the application handle any user input anyway. For this purpose, a special function called `e32.ao_yield()` is provided. By calling this function, you ensure that the user interface stays responsive although your application may perform computation in a tight loop at full speed.

Double buffering

Double buffering is a traditional method to prevent flickering when moving graphics are shown. This method was used to handle redrawing of the `Canvas` object.

As in the previous examples, instead of drawing the game elements to the `Canvas` object directly, we have a separate image object, `buf`, to which all elements are drawn at first. Only when everything has been drawn to the image buffer are the contents of the second buffer shown on the canvas using `canvas.blit(buf)`. This way the user does not have to see every individual drawing operation, but only the final result for each frame, which also reduces flickering on the screen.

Module `random`

Last but not least, many games need a random ingredient to stay interesting. Python provides a standard module called `random` that contains methods to generate random numbers. Here, the function `random.randint()` is used to generate integer values between the given minimum and maximum values. Another often used function, `random.random()`, returns a floating point value between 0 and 1.0.

5.5.2 The Game Application Code

The actual game code is divided into three parts that should be combined to make the final game. In the following discussion, we go through the listings one by one.

Example 37: UFO Zapper (1/3)

```
import key_codes, appuifw, e32, graphics, random

MAX_UFO_SIZE = 50
MAX_UFOS = 7
UFO_TIME = 100.0

PAD_W = 40
```

```
PAD_H = 15
PAD_COLOR = (12, 116, 204)
PAD_SPEED = 7
LASER_COLOR = (255, 0, 204)
TIME = 1000

def handle_redraw(rect):
    global shoot, sleep, ufos, timer
    buf.clear((0, 0, 0))
    buf.rectangle((pad_x, H - PAD_H, pad_x + PAD_W, H), fill = PAD_COLOR)

    if shoot:
        x = pad_x + PAD_W / 2
        buf.line((x, H - PAD_H, x, 0), width = 2, outline = LASER_COLOR)
        shoot = False
        sleep = 0.1
        check_hits(x)
    else:
        sleep = 0.01

    for x, y, s, t, hit in ufos:
        f = 1.0 - (timer - t) / UFO_TIME
        if hit:
            c = (255, 0, 0)
        else:
            c = (0, f * 255, 0)
        buf.ellipse((x, y, x + s, y + s), fill = c)

    buf.text((10, 40), u"%d" % score, fill = LASER_COLOR, font = "title")
    buf.text((W - 70, 40), u"%d" % (TIME - timer),
        fill = LASER_COLOR, font = "title")

    canvas.blit(buf)
```

The first part of the game code in Example 37 declares the constants needed in the game:

- `MAX_UFO_SIZE` gives the maximum size of a UFO in pixels.
- `MAX_UFOS` controls how many UFOs are shown at once on the screen.
- `UFO_TIME` controls for how long a UFO stays on screen.
- `PAD_W` and `PAD_H` determine the pad size in pixels.
- `PAD_COLOR` determines color of the pad.
- `PAD_SPEED` determines how many pixels the pad advances in one frame.
- `LASER_COLOR` is the color of the laser beam.
- `TIME` specifies for how many frames the game is played. This is approximately `TIME * 0.05` seconds.

The function `handle_redraw()` is responsible for drawing the game on screen based on the internal state of the game. First, the previous frame is cleared and the pad is drawn to its current location on the X axis. If the player has activated the laser beam with the Select key, the beam is drawn and the function `check_hits()` is called to determine whether any UFOs have been hit. Then the laser beam is switched off again with `shoot = False` and the length of the pause (`sleep`) is temporarily increased to a tenth of a second so the beam and possible hits can be seen easily.

After this, the UFOs are drawn based on the list `ufos`, which contains the coordinates, size and lifetime of each UFO. The shade of a UFO is determined according to its age: the closer the age is to its maximum lifetime `UFO_TIME`, the darker shade of green is selected. However, if the UFO has been hit, as determined by the `check_hits` function above, it is painted in glowing red. Finally, the current score and time are drawn on screen.

Example 38: UFO Zapper (2/3)

```
def check_hits(laser_x):
    global ufos, score
    i = 0
    ok_ufos = []
    for x, y, s, t, hit in ufos:
        if laser_x > x and laser_x < x + s:
            ok_ufos.append((x, y, s, t, True))
            score += MAX_UFO_SIZE - (s - 1)
        else:
            ok_ufos.append((x, y, s, t, False))
    ufos = ok_ufos

def update_ufos():
    global ufos, timer
    ok_ufos = []
    for x, y, s, t, hit in ufos:
        if not hit and timer < t + UFO_TIME:
            ok_ufos.append((x, y, s, t, False))
    ufos = ok_ufos

    if len(ufos) < MAX_UFOS:
        s = random.randint(10, MAX_UFO_SIZE)
        x = random.randint(0, W - s)
        y = random.randint(0, H - PAD_H * 3)
        t = random.randint(0, UFO_TIME)
        ufos.append((x, y, s, timer + t, False))
```

Example 38 lists the two functions related to the maintenance of the armada of UFOs. The `check_hits()` function checks which of the UFOs, if any, have been hit by the laser beam at the X coordinate, `laser_x`. This is accomplished by checking whether any of the UFO

circles intersect with the beam. If the beam touches a UFO, we mark the hit entry in the corresponding UFO tuple as `True`.

The function `update_ufos` is responsible for cleaning up any UFOs that have been shot down (that is, `hit == True`) or which have become too old. If the number of UFOs on screen is less than `MAX_UFOS` after this process, a new UFO of random size is created at a random location.

Example 39: UFO Zapper (3/3)

```
def handle_event(event):
    global direction, shoot
    if event['keycode'] == key_codes.EKeyLeftArrow:
        direction = -PAD_SPEED
    elif event['keycode'] == key_codes.EKeyRightArrow:
        direction = PAD_SPEED
    elif event['keycode'] == key_codes.EKeySelect:
        shoot = True

def quit():
    global timer
    timer = TIME

ufos = []
shoot = False
direction = pad_x = score = timer = 0
appuifw.app.exit_key_handler = quit
appuifw.app.screen = "large"
canvas = appuifw.Canvas(event_callback = handle_event,\
                        redraw_callback = handle_redraw)
W, H = canvas.size
buf = graphics.Image.new((W, H))
appuifw.app.body = canvas

while timer < TIME:
    pad_x += direction
    pad_x = min(pad_x, W - PAD_W)
    pad_x = max(pad_x, 0)

    update_ufos()
    handle_redraw((W, H))
    e32.ao_sleep(sleep)
    timer += 1

print "Your final score was %d!" % score
```

The final part of the game, listed in Example 39, includes the simple function `handle_event()` which registers any left or right key presses and changes the direction of the pad accordingly. If the user clicks the Select key, the laser beam is activated by setting the global variable `shoot = True`.

The game code ends with the event loop. The pad is advanced the amount determined by the constant `PAD_SPEED`, however keeping it within the screen limits. After this, status of the UFOs is updated. Then

the screen is re-drawn and the frame is paused on screen for a `sleep` fraction of a second. The `e32.ao_sleep()` function lets any new user events, such as key presses, be processed during the pause. Finally, the time counter, `timer`, is incremented by one and the event loop starts again from the beginning.

This example, besides being a highly addictive game, introduced several useful concepts that can be re-used in many game-like applications. As a further exercise, you might want to add background music and sound effects to make the game even more engaging. Creating a high-score list might not be a bad idea, especially after you have read Chapter 6, which explains how to load and save data.

5.6 Summary

In this chapter we first showed how to program applications to play and record sounds in various formats. Then, we described three approaches to programming keyboard keys and we outlined how to draw graphics primitives to the screen, with other useful features of the `graphics` module. We introduced the basic functionalities of the phone's internal camera and showed the basic principles of making simple games.

We hope you are inspired by the plethora of features that PyS60 offers regarding sound, camera and interactive graphics. In this chapter, we introduced each of these features separately, using rather simplistic, but hopefully illustrative examples. However, in reality, we think these features are best served as an unprecedented, rich and smooth mixture, preferably combined with other ingredients that are described in the following chapters.

6
Data Handling

When building applications, sooner or later you will come to the point where you want to store some user settings, log data, videos, music, or image files to the phone. This chapter gives you an introduction to handling data like this. As you will see, PyS60 is quite flexible and easy to use in this regard.

Nowadays, data persistence is often handled on the server side. Think of Google GMail, Flickr or the contact list in Skype – all these services ensure that your information moves with you even though you may access the services from various physical devices. Backups, scalability and data aggregation are easier to handle on the server side than on a single small device. These viewpoints should encourage you to think about storing data in unorthodox ways when programming mobile applications with PyS60. In many cases, sending data to a server is easier than saving it locally.

In some cases, saving data locally is the only sensible option. You should be able to save photos, sounds or log events without a network connection. You may need a local configuration file to specify how to setup a network connection in the first place. And since mobile networks may be unreliable and often have low bandwidth, caching information locally can be vital.

This is where the lessons of this chapter prove useful. First, in Section 6.1, we explain where files can be loaded and saved on the S60 platform. We introduce the `File` object that handles reading and writing of files. In Section 6.2, we present a slightly different approach that is based on the phone's built-in database engine. Then, in Section 6.3, we present a brief overview of current positioning techniques and how to use them in PyS60.

To show how to apply these techniques in practice, this chapter includes two useful applications: in Section 6.3, we build an application that tracks your location based on GSM cell IDs, which demonstrates file access and a positioning technique. Finally, in Section 6.5, we show

how you can combine the camera, a sound recorder and some text input dialogs to create your own vocabulary-learning tool for your next holiday in a foreign country.

This chapter also includes two important Python language lessons: one about exception handling and the other about a data structure called dictionary.

6.1 File Basics

Files on your Symbian device are organized in a hierarchical manner as drives and directories, similarly to a typical Windows system. The drive letters refer to the following resources:

- C: internal memory of your device
- D: operating memory space or RAM (read-only)
- E: memory card
- Z: fixed memory space or ROM (read-only).

You don't need to worry about drives D: and Z: since they are used by the operating system only. Starting with Symbian OS v9.0 (3rd Edition of S60), the directory hierarchy within the drives is strictly defined and only partly accessible to your program – see Appendix A for details.

Many examples in this book save their files to the memory card, that is, to the E: drive. If you do not have a memory card, you should change the drive letter to C:.

If your application must save anything to a file, such as private data, photos or sounds, it is polite to place the file in a logical place and try to avoid cluttering arbitrary directories unnecessarily.

If a file, typically a photo or a sound, should be visible to the standard Gallery or to other applications, including the Nokia PC Suite file manager, the standard `C:\Images` or `C:\Sounds` directories, or the corresponding locations on the E: drive, are appropriate choices.

If a file is private to your application and does not have to be visible to the user or any other application, you should make a private directory either under `C:\Data` or `E:\Data`.

The standard directories `E:\Images` and `E:\Sounds` are created by the built-in camera application and sound recorder, when you save a photo or a sound to the memory card for the first time. Thus, these directories are often already available for your application.

In contrast, any specific directory for your application has to be created before any files can be saved. The module `os` provides the function `os.makedirs()` that is used to create directories (see Example 40).

Example 40: Creating a directory for application data

```
import os, os.path
PATH = u"C:\\Data\\MyApp"
if not os.path.exists(PATH):
    os.makedirs(PATH)
```

We need to check that the directory does not exist before calling `os.makedirs()`, as it would throw an exception if the directory existed.

6.1.1 Handling Error Conditions

Especially when handling files, Bluetooth and network connections in the following chapters, exceptions are common. Often, you can design your program so that, say, a missing file or an already existing one do not cause a fatal error. Instead, your program could detect the situation and act accordingly.

It is not always possible or practical to detect the situation beforehand, as we did with the `os.path.exists()` function above. However, the program could *react* to the situation *after* an exception has occurred. Python provides a mechanism to detect and handle exceptional situations like this, which is introduced in the language lesson.

Python Language Lesson: catching exceptions

In Python, error conditions are objects, too. Whenever something unexpected happens, for example, you try to use a return value of a dialog even though it is `None`, an `Exception` object is created. You can create exceptions of your own in the following way:

```
raise Exception("I am exceptional")
```

If the exception is not handled by the program, it is shown on the console and execution of the program may terminate.

However, some exceptions are not really exceptional. For example, we know that trying to play a sound that does not exist will fail. It makes sense to prepare for exceptions like this. Exceptions are handled by enclosing suspicious actions between `try` and `except` statements, as in the following:

```
import appuifw
try:
    d = appuifw.query(u"Type a word", "text")
    appuifw.note(u"The word was: "  + word)
except:
    appuifw.note(u"Invalid word", "error")
```

If you type a word in the first dialog, everything works fine. However, if you click 'Cancel', an exception is raised since you cannot concatenate `None` to a string. This exception is handled inside the `except` block – in this case, an error dialog is shown.

To summarize, if you know that some operations are likely to cause exceptions and you know how to recover from them, place the suspicious operations between `try` and `except` statements and place alternative code in the `except` block.

However, use these statements sparingly, since they may also catch and hide exceptions that are caused by real errors in your program. The best approach would be to capture only specific types of exception, instead of all exceptions as we did above. See Python documentation for more information about this possibility.

Note that, because of space constraints, examples in this book do not always contain the `try–except` block around critical operations, even though using one might make sense. Feel free to make the examples more robust by adding more exception handlers that take care of possible error conditions properly.

6.1.2 File Object

In Python, files are read and written through the `File` object. Creating a `File` object opens a file, either for reading or writing. The object provides various functions for accessing data in the opened file.

The `File` object maintains a cursor that points at a specific location in the file, depending on what you have read or written most recently. Every time you read or write data to the file, the cursor is moved accordingly. As the cursor moves forward automatically, you can read through the file only once without explicitly repositioning or re-opening the file.

Similarly, once you have written data to a file through the `File` object, you cannot read the newly written data from a file unless you reposition or re-open the object. Conceptually, the cursor in the `File` object works similarly to the cursor in your text editor.

Here is a simple example that opens a file, writes a string to it and reads the string from the file. After you have executed the script you should see a new file, `C:\python\test.txt`, on your mobile device.

Example 41: Basic file operations

```
f = file(u"c:\\python\\test.txt", "w+")
print >> f, "Ip dip, sky blue"
f.seek(0)
print "File says", f.read()
f.close()
```

The first line opens the file. The file name must be specified as a Unicode string. Note that the path names must be separated by double backslashes since a single backslash marks a special character in strings. The second parameter denotes the mode in which the file is opened. Here are the options:

- `w` – the file is opened for writing. If the file already exists, it will be emptied first. Otherwise a new file is created.
- `r` – the file is opened for reading in text mode. An exception is generated if the file does not exist.
- `a` – the file is opened for writing. Writing continues at the end of file, if the file already exists. Otherwise a new file is created.

If a plus sign is added to the mode specifier, the file is opened in read–write mode, so you can both read from and write to the file using a single file object. The difference between 'w+' and 'r+' is that, in the former case, a new file is created if the file does not exist.

The easiest way to write anything to a file is to use the `print` statement. In contrast to the ordinary `print` statement, you need to specify a target for printing, which is given as a file object preceded by double-arrows, >>:

```
print >> f, "Ip dip, sky blue"
```

Note that the `print` statement always adds a new line character after the written string. If you want to avoid this, you can use the file object's `write()` function. In the example above, we could have written:

```
f.write("Ip dip, sky blue")
```

To read the newly created file, the file cursor must be repositioned back to the beginning. The function `seek()` performs this job. The value '0' specifies that the cursor is repositioned back to the beginning of the

file, which is the most typical use of the function. For other uses, see the Python documentation.

There are three main ways to read text data from a file:

- all at once: `txt = f.read()`
- one line: `line = f.readline()`
- a line at time: `for line in f: print line`

There are also some other ways to read data which are described in the Python documentation. In Example 41, everything in the file is read at once and then printed to the console.

The file is closed automatically when the file object is destroyed. For instance, this happens when you return from the function in which the file object was created, unless the variable holding the file object is defined as global.

Note that only after closing the file can you be sure that its contents are visible to other programs, as the file object may buffer data internally. To be on the safe side, you should call the `close()` function explicitly when you do not need the file, as we did in Example 41.

6.1.3 Logging to a File

In some cases, it is not feasible to leave the debugging output on the console. This might be the case if you need to output many debugging lines or you are using the standard application UI framework which hides the console output behind the application body.

In cases like this, it is a good idea to redirect the debugging output to a file. This is remarkably simple: just replace all `print` statements with `print >> logfile` statements, where `logfile` is a variable containing a file object that has been opened, as in Example 41.

If you are field-testing your program with, say, ten users, it makes sense to log all debugging events in a file. After the test session has ended, you can collect the log files and analyze the results. If you need a truly sophisticated solution, you can send the log files automatically to your server over a network connection. You can do this easily using techniques that are presented in Chapter 8.

6.1.4 Finding Sound, Photo and Video Files

The phone's built-in camera application saves photos and videos and the sound recorder saves sounds to a convenient location in the directory hierarchy of the phone, especially if you have set them to use the memory card. Note that some S60 2nd Edition devices may save files to other locations as well. Examples 42, 43 and 44 read files from the standard locations.

Videos are found in 3rd Edition phones in the directory named Videos. The phone automatically creates a new subfolder for each month and year when shooting a video. Example 44 reads a video that was recorded in March 2007.

Example 42: Read a sound

```
f = file("E:\\Sounds\\Digital\\boo.wav", "r")
mysound = f.read()
f.close()
```

Example 43: Read an image

```
f = file("E:\\Images\\picture.jpg", "r")
img = f.read()
f.close()
```

Example 44: Read a video

```
f = file("E:\\Videos\\200703\\video.mp4","r")
myvideo = f.read()
f.close()
```

6.2 Reading and Writing Text

If you need to read only one string from a file, say the user's name, calling `read()` once is enough, as it reads the whole file at once. But what if you need to read a list of strings instead?

We could save the list items one to a line and read the file one line at time. Note that list items are not allowed to contain line breaks in this case. Example 45 shows how to do this.

Example 45: Read and write text

```
def save_list(filename, lst):
    f = file(filename, "w")
    for item in lst:
        print >> f, item
    f.close()

def load_list(filename):
    f = file(filename, "r")
    lst = []
    for line in f:
        lst.append(line.strip())
    f.close()
    return lst
```

```
test_list = ["first line", "second line", "that's all"]
save_list(u"c:\\python\\test.txt", test_list)
print load_list(u"c:\\python\\test.txt")
```

This script should print out the contents of the list `test_list` when executed. The function `save_list()` writes each item of the given list to a file. The function `load_list()` loads items from the file, reading a line at time in a loop, and appends each line to the list `lst`. Note that we use the string function `strip()` to remove the trailing new line characters from lines that are read from the file. The list read from the file is identical to the original list `test_list`. A simple format like this might be enough for saving, say, a list of URL bookmarks.

6.2.1 Key–Value Pairs

What if a plain list of strings is not enough? For example, a configuration file might contain several different fields, such as a user name, password, server name and port, all of which need to be accessed separately. A plain list would not suffice, since we cannot know which line corresponds to which field.

In cases like this, key–value pairs come in handy. Python has a useful data structure for accessing key–value pairs, namely the dictionary, that was briefly mentioned in Section 5.2. The language lesson describes the use of the dictionary.

Python Language Lesson: dictionary

The dictionary object is used to save unique key–value pairs. It is a versatile data structure which is used in Python for many different purposes.

You can create a new dictionary as follows:

```
a = {}
b = {"state": "CA", "zip": 94301}
```

Here `a` contains an empty dictionary, defined with an empty pair of curly brackets. The variable `b` contains a dictionary that is initialized with two key–value pairs: it contains the keys ‘state’ and ‘zip’ and the corresponding values ‘CA’ and ‘94301’.

Note that values can be any type of object, even `File` objects or images, but key types have some restrictions. Simple types, such as strings and numbers, may be used as keys. Values are accessed with keys as follows:

```
print b["state"]
b["state"] = "NY"
print b["state"]
b["newkey"] = "new value"
print b["newkey"]
```

This above code prints out 'CA' followed by 'NY' and 'new value'. You may change values and add new key–value pairs to the dictionary freely after it has been created. However, a key can map to only one value, so assigning a new value to an existing key replaces the previous value. If you want to have multiple values per key, use a list as the value and append new items to it.

A key–value pair can be deleted as follows:

```
del b["state"]
```

You can test if a key exists in a dictionary as follows:

```
if "state"in b:
    print "state exists in the dictionary!"
else:
    print "state does not exist in the dictionary"
```

You can loop through all keys and values in the dictionary as follows:

```
for key, value in b.items():
    print "KEY", key, "VALUE", value
```

The number of keys in the dictionary can be found with `len(b)`. The dictionary supports a number of other operations as well. See the Python documentation for more information.

6.2.2 Reading and Writing Named Values

Using dictionaries, we can develop functions for saving and loading a configuration file that consists of keys and values. The format presented here is easy to read and edit manually. A downside is that only strings are accepted as keys and values. Also, the saved strings must not contain any new line characters or colons. In Section 6.3, you learn a more sophisticated way for saving information like this to a local database.

Writing contents of a dictionary to a file is straightforward using the concepts that we have just learnt.

Example 46: Writing a dictionary to a file

```
def save_dictionary(filename, dict):
    f = file(filename, "w")
    for key, value in dict.items():
        print >> f, "%s: %s" % (key, value)
    f.close()
```

One key–value pair is saved per line. The key and value strings are separated by a colon character. Because of these choices, the strings cannot contain any colons or new line characters.

A saved configuration file might look like this:

```
Host: www.google.com
Port: 80
Username: my_login
Password: this is a visible secret
```

Reading the file is a bit trickier than writing it.

Example 47: Read a dictionary from a file

```
def load_dictionary(filename):
    f = file(filename, "r")
    dict = {}
    for line in f:
        key, value = line.split(":")
        dict[key.strip()] = value.strip()
    f.close()
    return dict
```

We go through the file line by line. We split each line into the key part and the value part, using the colon as a separator. Then, the new key and value are updated to the dictionary `dict`. As before, we use function `strip()` to get rid of all leading and trailing white space that might occur in keys and values.

6.2.3 Reading and Writing Unicode Text

Both Symbian OS and the S60 platform are designed for world-wide use. Consequently, every part of the system that deals with text in any form, is designed to handle text written in any language, using any character set.

Unicode is a standard that defines how the world's writing systems should be handled on computers. Unicode is natively supported by the Symbian and S60 platforms, as well as by PyS60. As you have seen, practically all functions in the PyS60 API that handle strings require that the input is specified as Unicode strings. In Python, Unicode strings can be recognized by the `u` prefix before the quotes, as in `u"Kanji"`.

However, Python functions that are not part of the mobile platform and PyS60 API, but parts of the Python standard library, often do not handle Unicode automatically. For instance, `File` objects and network connections do not require that the data must be specified in Unicode. As a consequence, you have to handle conversions between plain strings and Unicode strings explicitly in your code when using these functions, as we describe below.

In the following code, the first line converts a Unicode string `orig_str` to the plain string `plain_str` and the second line converts it back to Unicode.

```
orig_str = u"käärmekännykkä"
plain_str = orig_str.encode("utf-8")
unicode_str = plain_str.decode("utf-8")
```

There are many ways to encode a Unicode string to a plain string, but here we use an encoding called UTF-8 that is widely used nowadays.

Generally speaking, it is enough to remember the following rule of thumb: whenever you save a string from PyS60 to a file, use `encode("utf-8")` before writing. Correspondingly, when you read the string from a file, use `decode("utf-8")` before using the string in your application. The same rule applies also for the local database, as well as for Bluetooth and network communication.

In this book, we have omitted the conversion in some examples for brevity. All the larger application examples should perform the conversion whenever it is needed.

6.3 Local Database

Symbian OS includes a relational database engine, to which PyS60 provides two interfaces. The first interface can be found in module `e32db` and the second interface in module `e32dbm`.

The former interface provides low-level, versatile access to the database. It supports transactions and querying the database with a subset of Simple Query Language (SQL). This interface can be useful for applications that need to store relationally organized data and perform queries frequently. In this case, having a local database, instead of sending data to a 'real' database on a server, may be justified.

In many cases, the interface provided in module e32dbm is preferred because of its simplicity. In essence, you can treat e32dbm as a persistent dictionary for string keys and values. It supports most of the standard dictionary functions, such as adding, deleting and iterating through key–value pairs.

Example 48: Local database

```
import e32dbm

DB_FILE = u"c:\\python\\test.db"

def write_db():
    db = e32dbm.open(DB_FILE, "cf")
    db[u"host"] = u"www.google.com"
    db[u"port"] = u"80"
    db[u"username"] = u"musli"
    db[u"password"] = u"my secret"
    db.close()

def read_db():
    db = e32dbm.open(DB_FILE, "r")
    for key, value in db.items():
        print "KEY", key, "VALUE", value
    db.close()

print "Writing db.."
write_db()
print "Reading db.."
read_db()
```

As you can see, the local database module, e32dbm, behaves like a mixture of a file and a dictionary: it is opened like a file and accessed like a dictionary. Note that the database has native support for Unicode strings, so no conversions have to be performed.

You need to specify the file name where the database is to be stored. The mode parameter works differently from that of the File object:

- r – opens the database for reading only.
- w – opens the database for reading and writing.
- c – opens the database for reading and writing (and creates a new database if the file does not exist).
- n – opens the database for reading and writing (it creates an empty database if the file does not exist or clears the previous database).

If you add f after the mode character, the database is not updated on disk until you close it or force it to be written to disk by using a special function. This makes the database access faster. You should use this mode unless you need the additional safety of synchronized writing.

Given the convenience of a local database, why should one ever consider using an *ad hoc* solution as in Examples 45 and 46? The biggest reason is interoperability. You can read and write text files on any device, including your PC and a server running, for instance, FreeBSD. On the other hand, databases created by the `e32dbm` interface are strictly Symbian-specific. However, if the data is only to be accessed privately by a PyS60 application, the local database is probably a safe choice.

For a practical application example of the local database, see Section 9.3.

6.4 GSM and GPS Positioning

Applications based on positioning are often claimed to be the most likely killer applications of the mobile platform. PyS60 gives you access to practically all public positioning techniques that are available for the current S60 mobile phones. As of 2007, the most typical positioning techniques are the following:

- external GPS over Bluetooth, as described in Section 7.5
- internal GPS, using the `position` module
- GSM cell IDs, using the `location` module
- SMS-based positioning using the `messaging` module. This service is provided by some mobile phone operators.

Besides these techniques, various positioning techniques based on WiFi or Bluetooth have been experimented with by research groups using PyS60 and custom extensions.

On S60 3rd Edition phones, access to both the internal GPS and GSM cell IDs are restricted by additional capabilities, which are not available for self-signed PyS60 – see Appendix A for more information. This means that if you have an S60 3rd Edition phone, you cannot use Example 49 with the self-signed PyS60 interpreter that we installed in Chapter 2. Instead, you need to obtain a personal, free, developer certificate from Symbian and sign the interpreter with it, as described in Section A.4.3. On S60 2nd Edition devices, no platform security is enforced and the GSM cell IDs can be accessed without trouble.

6.4.1 GSM Cell ID Mapper

We present a simple application that can be used to record positioning data and to retrieve your current location, based on GSM cell IDs. The example is based on the GSM cell IDs because of the ubiquity of the

technique. Only a few phones have an internal GPS receiver (as of 2007) but every GSM phone can be used to retrieve GSM cell IDs.

This application allows you to collect cell IDs and give them names according to your actual location. The mapping from the detected cell IDs to their locations is saved in a text file. This file can be edited and updated, for instance, based on publicly available information about the locations of GSM cells. In addition, the application logs your current location to a file with a timestamp, so you can analyze your movements afterwards.

The core functionality of Example 49 is based on the module `location`.

Example 49: Retrieve the current GSM cell ID

```
import location
print location.gsm_location()
```

You should see a tuple of four integers whose values depend on your current location. The first two integers correspond to the Mobile Country Code (MCC) and Mobile Network Code (MNC). A comprehensive mapping of MCC and MNC codes to country and operator names can be found in the Wikipedia article 'List of GSM Network Codes'.

The last two integers correspond to the GSM area code and the current GSM cell ID. Many different cell IDs may map to a single area code, which, in urban environments, may cover a district.

Note that you are likely to see many different cell IDs in one physical location, at least if you test the system on the same location at different times. This is because of overlapping cells and other intricacies of the GSM network.

Thus, even though you may have named a certain location earlier, there is no guarantee that the same location will be recognized afterwards when you arrive at it again. However, the more cells you name the more probable it is that some mappings will eventually match.

Example 50 contains the source code for the GSM cell ID mapper. This example uses the functions `save_dictionary()` and `load_dictionary()` that are found in Examples 46 and 47. To save space, these functions are not repeated here, but you must include them in the actual application.

Example 50: GSM location application

```
import appuifw, e32, location, time, os.path

PATH = u"E:\\Data\\gsm_loca\\"
if not os.path.exists(PATH):
    os.makedirs(PATH)
```

```
INTERVAL = 5.0
CELL_FILE = PATH + "known_cells.txt"
LOG_FILE = PATH + "visited_cells.txt"
log = file(LOG_FILE, "a")
timer = e32.Ao_timer()

def current_location():
    gsm_loc = location.gsm_location()
    return "%d/%d/%d/%d" % gsm_loc

def show_location():
    loc = current_location()
    if loc in known_cells:
        here = known_cells[loc]
        print "You are currently at", here
    else:
        here = ""
        print "Unknown location", loc

    print >> log, time.ctime(), loc, here
    timer.after(INTERVAL, show_location)

def name_location():
    loc = current_location()
    name = appuifw.query(u"Name this location", "text")
    if name:
        known_cells[loc] = name

def load_cells():
    global known_cells
    try:
        known_cells = load_dictionary(CELL_FILE)
    except:
        known_cells = {}

def quit():
    print "SAVING LOCATIONS TO", CELL_FILE
    save_dictionary(CELL_FILE, known_cells)
    print "GSM LOCATIONING APP EXITS"
    timer.cancel()
    log.close()
    app_lock.signal()

appuifw.app.exit_key_handler = quit
appuifw.app.title = u"GSM location App"
appuifw.app.menu = [(u"Name this location", name_location)]

print "RECORDING VISITED CELLS TO", LOG_FILE
print "LOADING LOCATIONS FROM", CELL_FILE
load_cells()
print "%d KNOWN CELLS LOADED" % len(known_cells)
show_location()

print "GSM LOCATIONING APP STARTED"
app_lock = e32.Ao_lock()
app_lock.wait()
```

When the application starts, we try to load previously saved locations from the file `CELL_FILE` in the function `load_cells()`. Here we use a `try-except` block: if the file does not exist, `load_dictionary()` throws an exception which is caught by the `except` statement. In this case, we start with an empty dictionary.

The core data structure is the dictionary `known_cells`. It contains a mapping from the known GSM cell IDs to their names. It is initialized by the function `load_cells()`, as mentioned above. When the user chooses 'Name this location', we update the dictionary in the function `name_location()`. When the application quits, the dictionary is saved to the file `CELL_FILE` using the function `save_dictionary()`.

As this application is used to track the user's movements, we need to check the current location periodically. The module `e32` provides a timer object called `e32.Ao_timer` that can be used to call a function after a certain number of seconds has elapsed. In this case, the expression `timer.after(INTERVAL, show_location)` specifies that the function `show_location()` should be called after `INTERVAL` seconds. The `after()` function puts the request on hold and returns immediately. For another example of the `Ao_timer` object, see Section 9.3.

The result of the timer operation is that the function `show_location()` is called after the specified interval. This function checks and reports your current location, based on the `known_cells` dictionary, and logs your current location using the file object `log`, which is opened when the application starts. Also the current time, as returned by the function `time.ctime()` is added to each location logged. Before the function `show_location()` returns, it sets the timer to call itself again after the interval, so the location is checked periodically.

The function `current_location()` is responsible for fetching the actual GSM cell ID. It works in a similar way to Example 49 above. Here, however, we convert the result to a string, so it can easily be printed out on the screen and saved to a file.

Note that, when the user quits the application, we have to cancel any pending timer request using the function `timer.cancel()`. Otherwise, the request could activate after the application has already quit, causing the PyS60 interpreter to crash.

Once you have the application running, you should take a trip around your neighborhood. First, try to find different cells. Once you have spotted two or three cell IDs in different physical locations, give a name to each of them. Walk the same route again and see if the names shown on the screen correspond to the real locations.

Do not be surprised if the cells seem to overlap or disappear, or new cells appear although you stay in one location. In reality, building a really accurate positioning system solely based on GSM cell IDs is a major challenge.

6.4.2 GPS Positioning

The PyS60 API documentation contains more information, including some examples, about the usage of the `position` module, which relies on the internal GPS. This module should be of interest to those who own a phone, such as Nokia N95, that includes a built-in GPS receiver. Note that also in this case you need a signed PyS60 interpreter, as described above.

6.5 Vocabulector: A Language-Learning Tool

The final example of this chapter is a personal language-learning tool. Let's assume you are traveling in the Basque country and you want to buy some cheese. You may find in your dictionary that cheese is 'gazta' in Basque, but you have no idea how to pronounce it correctly. Probably you would forget the word anyway before you get to a local cheese-shop.

Vocabulector solves the problem. In your hotel room, you type 'cheese' as the native word and 'gazta' as the foreign word into your Vocabulector and take a photo of a half-eaten piece of cheese. After this, you find a friendly person on the street, show her the photo and ask her to pronounce it like a native.

The correct pronunciation is recorded by Vocabulector and saved with the two words and the photo. Once you get to the cheese shop, you have the correct pronunciation and a descriptive photo readily available. Not only do you get the cheese, but you can practice pronunciation and hearing comprehension with this handy little application afterwards in your hotel room, while eating your *gazta*.

This example relies heavily on the ability to load and save data. Words, photos and sounds are saved to an application-specific directory at `E:\Data\Vocabulector`. Besides loading and saving files, this example also uses many concepts from Chapter 5, most notably the camera and sounds.

The code is divided into three parts. Example 51 deals with adding new entries to Vocabulector and loading any previously saved ones. Example 52 is responsible for showing the entries and Example 53 contains the standard application UI boilerplate code. You should combine these parts together in one file to form the full application.

6.5.1 Adding New Entries to Vocabulector

Example 51: Vocabulector (1/3)

```
import os, os.path, audio, appuifw, camera, key_codes, graphics
```

```
PATH = "E:\\Data\\Vocabulector\\"

def load_translations():
    global trans, dict_file
    if not os.path.exists(PATH):
        os.makedirs(PATH)
    dict_file = file(PATH + "trans.txt", "rw")
    trans = []
    for line in dict_file:
        line = line.decode("utf-8")
        native, foreign = line.strip().split(":")
        trans.append((native, foreign))

def add_entry():
    global text, photo, fname
    text = photo = None
    native = appuifw.query(u"Native word:", "text")
    if not native:
        return
    foreign = appuifw.query(u"Foreign word:", "text")
    if not foreign:
        return
    fname = PATH + native
    trans.append((native, foreign))
    line = "%s:%s" % (native, foreign)
    print >> dict_file, line.encode("utf-8")

    if appuifw.query(u"Record sound", "query"):
        record_sound()
    if appuifw.query(u"Take photo", "query"):
        camera.start_finder(viewfinder)
        canvas.bind(key_codes.EKeySelect, take_photo)
    else:
        appuifw.note(u"Entry added!", "info")

def record_sound():
    snd = audio.Sound.open(fname + ".wav")
    snd.record()
    appuifw.query(u"Press OK to stop recording", "query")
    snd.close()

def viewfinder(img):
    canvas.blit(img)

def take_photo():
    global photo
    canvas.bind(key_codes.EKeySelect, None)
    camera.stop_finder()
    photo = camera.take_photo(size = (640,480))
    handle_redraw(None)
    photo.save(fname + ".jpg")
```

The constant `PATH` defines a directory for the Vocabulector's files. If you do not have a memory card, replace E: in the path with C: to refer to the phone's internal memory. The function `load_translations()` loads the saved word pairs from a text file when the application starts. The file is opened in `rw` mode, meaning that we may read from and write

to the file using the same `File` object `dict_file`. A new file is created if it does not exist already.

Each line in the `trans.txt` file contains a pair of words separated by a colon, for instance, `cheese:gazta`. Since the application deals with foreign words, we have to encode Unicode strings properly in UTF-8 before writing them to a file. Correspondingly, the UTF-8-encoded lines that are read from the file must be decoded back to Unicode strings. The core data structure is the list `trans` that contains the list of native–foreign word pairs, saved in tuples.

The function `add_entry()` is invoked from the application menu. It asks for a new native word and the corresponding word in a foreign language. The new pair is then appended to the file using the `dict_file` file object. If the user can provide a sound or a photo to accompany the new word, these files are named according to the native word by adding a `.jpg` or a `.wav` extension to the base file name, `fname`.

Recording sound is a straightforward operation, as can be seen in the function `record_sound()`. Taking a photo is not much more difficult: once the `camera.start_finder()` call has switched the viewfinder on, the function `viewfinder()` starts showing frames from the camera on the canvas. We also bind the select key to the function `take_photo()`. Once the key is pressed and `take_photo()` is called, we unbind the key so that only one photo can be taken at a time.

6.5.2 Showing Entries from Vocabulector

Example 52: Vocabulector (2/3)

```
def show_native():
    lst = []
    for native, foreign in trans:
        lst.append(native)
    idx = appuifw.selection_list(choices = lst, search_field = 1)
    if idx != None:
        foreign = trans[idx][1]
        fname = PATH + lst[idx]
        show(fname, foreign)

def show_foreign():
    lst = []
    for native, foreign in trans:
        lst.append(foreign)
    idx = appuifw.selection_list(choices = lst, search_field = 1)
    if idx != None:
        native = trans[idx][0]
        fname = PATH + native
        show(fname, native)

def show(fname, translation):
    global photo, text, snd
    photo = None
```

```
    text = translation
    try:
        photo = graphics.Image.open(fname + ".jpg")
    except: pass
    handle_redraw(None)
    try:
        snd = audio.Sound.open(fname + ".wav")
        snd.play()
    except: pass
```

The functions `show_native()` and `show_foreign()` are invoked from the application menu. The functions share a similar structure: they choose either the first or the second item, that is, either a native or a foreign word from the list of word pairs to show a list of words in a selection list dialog. Once the user has chosen a word from the list, the function `show()` is called; it loads the photo, plays the sound and requests the translation to be displayed on screen. Since both the photo and the sound may be missing, we ignore any exceptions that may occur while the files are being loaded.

In Python, statement `pass` means ‘no operation’. It is a syntactic placeholder saying that we have deliberately decided to do nothing if an exception occurs during loading photos or sounds. You could replace the `pass` statements with appropriate error messages, for example.

6.5.3 Vocabulector Boilerplate Text

Example 53: Vocabulector (3/3)

```
def handle_redraw(rect):
    canvas.clear((255, 255, 255))
    w, h = canvas.size
    if photo:
        canvas.blit(photo, target = (0, 0, w, int(0.75 * h)), scale = 1)
    if text:
        canvas.text((20, h / 2), text, fill = (0, 0, 255), font = "title")

def quit():
    dict_file.close()
    app_lock.signal()

photo = None
text = u"<vocabulector>"
appuifw.app.title = u"Personal vocabulary trainer"
appuifw.app.exit_key_handler = quit
appuifw.app.screen = "large"

canvas = appuifw.Canvas(redraw_callback = handle_redraw)
appuifw.app.body = canvas
appuifw.app.menu = [(u"Add entry", add_entry),
        (u"Show native entries", show_native),
        (u"Show foreign entries", show_foreign)]
```

```
load_translations()
app_lock = e32.Ao_lock()
app_lock.wait()
```

Similar to previous examples that have used the `Canvas` object, such as the UFO Zapper in Section 5.5, the drawing operations are handled in the redraw callback function `handle_redraw()`. The function is simple: if a photo is loaded, it is shown on the canvas. If the variable `text` contains a string, it is drawn on the screen as well. The rest of the lines correspond to standard UI boilerplate code, which should be familiar to you from previous examples.

6.6 Summary

In this chapter, we have gone through many new concepts, including:

- the directory hierarchy on the S60 platform
- making directories
- handling error conditions
- reading and writing files
- reading sounds, photos and videos
- the dictionary object
- handling Unicode conversions
- using a local database
- positioning techniques
- using a timer object.

By no means do you have to learn all these techniques at once. Feel free to use this chapter as a reference and come back to it whenever you need to apply these concepts in practice. Many of these ideas, such as reading and writing files and handling error conditions, are not specific to PyS60 but they apply to the Python language in general. Thus, you will find plenty of help on the web regarding these subjects.

Even though these techniques might not feel as intriguing at first sight as graphics and sounds, they form the basic scaffolding for any non-trivial application. Luckily, as you might have noticed, PyS60 makes this scaffolding extremely lightweight and transparent, so you can really focus on the things that are important to you.

7

Bluetooth and Telephone Functionality

The mobile phone is a strong candidate for becoming *the* interaction device that bridges the physical and virtual worlds. Standard networking techniques, such as TCP/IP, are great for heavy-duty communication over long distances. However, the last five meters between the phone and its physical surroundings are better handled with a lighter approach. In this range, Bluetooth is the dominant means of communication.

Lots of interesting things happen within that five-meter radius. You can use Bluetooth for social interaction by connecting to other phones and their users near to you. You can use it to interact with physical objects, such as public screens, GPS receivers, sensors, robotic vacuum cleaners and even shop windows. Naturally, Bluetooth also connects you to nearby PCs.

In this chapter, we explore most of these scenarios. We show how to send photos to and chat with other phones in Section 7.3. Then we get you started with phone-to-PC communication, which really becomes fun once you can also build applications on the PC side. For example, we show how to control your Apple Mac with your phone using AppleScript. In Section 7.5, we connect to an external GPS receiver and later, in Chapter 11, we connect to a sensor board. This might be of interest, for instance, to projects dealing with art installations, sensor networks and physical and wearable computing or smart fashion.

At the end of this chapter, we take a quick look at the telephone functionality of PyS60. Paradoxically, the modern mobile phone is a device of so many capabilities, that only a brief section is dedicated to its original purpose. Finally, we introduce a small but useful module, `sysinfo`, that contains lots of interesting trivia about the status of your phone.

7.1 Bluetooth Pairing

Security concerns mean that it is a good habit to *pair* any two devices before you connect them using Bluetooth. Pairing need be done only once for any two devices. The manual of your mobile phone contains instructions on how to do this. Typically, you can initiate pairing on the phone side in the Bluetooth configuration dialog.

The basic idea is that the phone shows a dialog asking you to type a passcode for the device – you can type any code you like. The other device should also show a dialog asking you to type the same passcode there as well. If your phone asks you to 'Authorize device to make connections automatically', you should answer 'yes' unless you want to authorize each connection individually, on a case by case basis.

7.2 OBEX and RFCOMM

There are two main ways to communicate over Bluetooth with PyS60. OBject EXchange (OBEX) is suitable for transferring files, such as photos or sounds, over Bluetooth. Radio Frequency COMMunication (RFCOMM) is useful for sending and receiving streams of text and raw data, including protocols of your own.

To send anything over Bluetooth, you have to know the recipient's *Bluetooth address*, which is represented in a string, such as `00:12:d2:41:35:e4`. You can find Bluetooth devices and their addresses with Bluetooth scanning. Typically, you have to worry about the raw addresses only if you want to connect to the same device again without scanning.

The Bluetooth address identifies a device. A single device may provide several *services*. For example, your phone can simultaneously receive files and communicate with a headset and a wireless keyboard over Bluetooth. Each of the services operates on a different *channel* so they do not interfere with each other. If you want to send a file to another phone, first you have to find out its address and then the channel that is used by the file transfer service.

This might sound a bit complicated and, in the background, it is. However, PyS60 wraps this functionality behind two simple functions: `socket.bt_obex_discover()` performs service discovery for the OBEX protocol and `socket.bt_discover()` performs service discovery for RFCOMM. Note that all Bluetooth-related functionality can be found in the `socket` module, which is described in detail in Chapter 8.

Note that to use these examples, you have to switch on Bluetooth on your phone. In many Nokia models, Bluetooth settings can be found under the Connectivity panel.



Figure 7.1 Bluetooth discovery

Example 54 performs Bluetooth scanning and shows a dialog such as the one in Figure 7.1, which lets the user choose a device to connect to. Once the user has made a choice, the function returns the Bluetooth address of the chosen device. It also returns a dictionary that lists available services on the device and the channels that correspond to them.

Example 54: OBEX discovery

```
import socket
address, services = socket.bt_obex_discover()
print "Chosen device:", address, services
```

The output of the example is something like this:

```
Chosen device: 00:12:d2:41:35:e4 {u"OBEX Object Push":9}
```

In this case, the target device, whose address is contained in the string `address`, has only one OBEX service available, `"OBEX Object Push"` on channel 9. This service is used to transfer files between devices. If no services were available on the target device, the function would have raised an exception. The service names are standardized, so if you are interested in only the `"OBEX Object Push"` service, you can request that particular key from the `services` dictionary.

If you know the recipient's Bluetooth address beforehand, you can find out the services provided by it without showing the dialog of Figure 7.1. The discover functions accept an address string as an optional parameter:

```
MY_PC = "00:12:d2:41:35:e4"
address, services = socket.bt_obex_discover(MY_PC)
```

We can discover available RFCOMM services in a similar manner. We just use the function `socket.bt_discover()` instead of `bt_obex_discover()`, as in Example 55.

Example 55: RFCOMM discovery

```
import socket
address, services = socket.bt_discover()
print "Chosen device:", address, services
```

Again, if the chosen device does not provide any RFCOMM services, the function raises an exception.

7.3 Phone-to-Phone Communication

Now that we have seen how to find a target for Bluetooth communication, we can start to do something useful. This section requires that you invite a friend with a Bluetooth-capable phone to hack with you. Bluetooth really enables social interaction.

7.3.1 Using OBEX

We build an application that lets you take photos and share them with people around you. Example 56 is a combination of the camera application of Example 35 and the OBEX discovery Example 54. The only

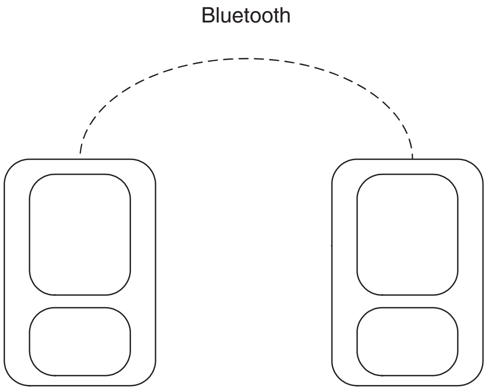


Figure 7.2 Bluetooth from phone to phone

new feature, sending a photo to a chosen recipient, can be performed with a single function, `socket.bt_obex_send_file()`.

Example 56: Send photos to another phone using Bluetooth

```
import camera, e32, socket, appuifw

PHOTO = u"e:\\Images\\bt_photo_send.jpg"

def send_photo():
    try:
        address, services = socket.bt_obex_discover()
    except:
        appuifw.note(u"OBEX Push not available", "error")
        return

    if u'OBEX Object Push' in services:
        channel = services[u'OBEX Object Push']
        socket.bt_obex_send_file(address, channel, PHOTO)
        appuifw.note(u"photo sent", "info")
    else:
        appuifw.note(u"OBEX Push not available", "error")

def take_photo():
    photo = camera.take_photo()
    canvas.blit(photo, scale = 1)
    photo.save(PHOTO)

def quit():
    app_lock.signal()

canvas = appuifw.Canvas()
appuifw.app.body = canvas
appuifw.app.exit_key_handler = quit
appuifw.app.title = u"BT photo send"
appuifw.app.menu = [(u"Take photo", take_photo),\
                    (u"Send photo", send_photo)]
app_lock = e32.Ao_lock()
app_lock.wait()
```

The application consists of two functions: `take_photo()` and `send_photo()`. The application structure should be familiar to you from the previous examples. Note that the viewfinder is not included in this example because of space constraints. You can easily add it by yourself, for instance based on Example 36.

To send the photo, we find the target device with `socket.bt_obex_discover()`. If the target supports the `OBEX Object Push` protocol, we find the channel that corresponds to it from the `services` dictionary. Once we know the recipient's address and the channel, we can send the photo to the recipient with `socket.bt_obex_send_file()`.

When the connection is established, the target device may show a popup asking 'Receive message via Bluetooth from...' – you should answer 'yes'. Since we use the standard `OBEX Object Push` service,

which is supported natively by the phone's operating system, the recipient does not need any special application to receive the photos. Photos appear in the recipient's standard inbox, in the same way as SMS or MMS messages.

7.3.2 Using RFCOMM

Example 56 relied on a standard OBEX service to transfer files from phone to phone, which is easy, but somewhat restricted. For instance, all data must be sent from a file and received into a file. It is also a connectionless protocol, which means that you cannot know whether the other side is still within your reach.

In contrast, RFCOMM opens a pipe between two devices, which can be used to send and receive any strings in your program – including files. In this respect, it is similar to a TCP/IP connection, so reading Chapter 8 should deepen your understanding of RFCOMM.

In this section, we create a simple chat application that lets you send short messages back and forth between two phones. Naturally, both phones need to be running the chat application. One phone must act as a server and the other as a client that initiates the connection.

After the client has established a connection to the server, the phones may start sending messages to each other in turn. Synchronous communication such as this is easier to handle than asynchronous communication, where the parties may send messages whenever they want. In Chapter 8, we show how to handle asynchronous communications as well.

The application is divided into two parts. You should combine the parts into one file to form the full application. Example 57 contains functions for both client and server. The `chat_server()` function waits for an incoming connection and `chat_client()` establishes a connection to the chat server.

Example 57: Bluetooth chat (1/2)

```
import socket, appuifw

def chat_server():
    server = socket.socket(socket.AF_BT, socket.SOCK_STREAM)
    channel = socket.bt_rfcomm_get_available_server_channel(server)
    server.bind(("", channel))
    server.listen(1)
    socket.bt_advertise_service(u"btchat", server, True, socket.RFCOMM)
    socket.set_security(server, socket.AUTH | socket.AUTHOR)
    print "Waiting for clients..."
    conn, client_addr = server.accept()
    print "Client connected!"
    talk(conn, None)

def chat_client():
    conn = socket.socket(socket.AF_BT, socket.SOCK_STREAM)
```

```
address, services = socket.bt_discover()
if 'btchat' in services:
    channel = services[u'btchat']
    conn.connect((address, channel))
    print "Connected to server!"
    talk(None, conn)
else:
    appuifw.note(u"Target is not running a btchat server", "error")
```

The function `chat_server()` may look a bit convoluted, but actually it just performs some housekeeping to prepare the system for clients to arrive. In other words, to be able to accept incoming Bluetooth connections, the server side must call the following functions:

1. `socket()` creates a new endpoint, or socket, for communication.
2. `bt_rfcomm_get_available_server_channel()` allocates a new channel for this service.
3. `bind()` binds the channel to the socket.
4. `listen()` informs the operating system that we are willing to accept incoming connections.
5. `bt_advertise_service()` makes our service, `btchat`, discoverable.
6. `set_security()` sets the security settings for this service.
7. `accept()` waits until a client establishes a connection with us.

Life is much easier for the clients, who only have to perform the following three steps in the function `chat_client()`:

1. `socket()` creates the endpoint for communication.
2. `bt_discover()` discovers the service of the other phone. The server must be started before the client, otherwise the client cannot find it.
3. `connect()` connects to the chosen server using the channel that is reserved for our `btchat` service.

Do not worry if the meaning of some of these steps is still unclear to you – the `socket` module is described more thoroughly in Chapter 8.

After these functions have been executed, `chat_server()` on one phone and `chat_client()` on the other, a new RFCOMM connection has been established between the phones. Both the functions end with a call to the `talk` function that orchestrates the actual chatting. Depending on whether `talk` is called from the server or the client side, the new

connection is passed to it either as a connection to the server or to the client.

Example 58: Bluetooth chat (2/2)

```
def receive_msg(fd):
    print "Waiting for message.."
    reply = fd.readline()
    print "Received: " + reply
    appuifw.note(unicode(reply), "info")

def send_msg(fd):
    msg = appuifw.query(u"Send a message:", "text")
    print "Sending: " + msg
    print >> fd, msg

def talk(client, server):
    try:
        if server:
            fd = server.makefile("rw", 0)
            receive_msg(fd)
        if client:
            fd = client.makefile("rw", 0)
        while True:
            send_msg(fd)
            receive_msg(fd)
    except:
        appuifw.note(u"Connection lost", "info")
        if client:
            client.close()
        if server:
            server.close()
        print "Bye!"

index = appuifw.popup_menu([u"New server", u"Connect to server"],
                                                   u"BTChat mode")
if index != None:
    if index:
        chat_client()
    else:
        chat_server()
```

When the `talk()` function is called from the `chat_client()` function, it gets a connection (socket object) to the server. In this case, the function starts waiting for the first message from the server side. On the server side, a dialog is shown which asks the user to type in the first message for the client.

The functions `receive_msg()` and `send_msg()` are used to receive and send messages. To use these functions, we convert the socket to a handy file-like object using `makefile("rw", 0)`. After this, we can read and write to the socket as if it was a normal file, in a similar way to what we did in Chapter 6. The second parameter, 0, tells that we do not need any buffering for communication.

This is the key to using RFCOMM: we can treat the Bluetooth connection as a file. However, care must be taken that we do not attempt to read from the connection-file if there is nothing to read, that is, the sender has not sent anything, and that we do not write too much to the connection until the reader has read at least some of it. In either case, the read or write function blocks and waits for the other side to act. If the other side does not act for some reason, the function waits forever and the program gets stuck.

Here we read and write only one line, or message, at a time. Sending is easy: it uses the normal `print` command, directed to the connection `fd`. Reading is performed with the file object's `readline()` function. Chapter 8 introduces other ways to transfer data over connections such as this.

The actual chatting takes place in an infinite `while` loop in the function `talk()`. We alternate between receiving a message and sending one. Since the server-side is the first one to enter the loop, it may also send the first message which lets the client into the loop as well. This alternate communication continues until some exception occurs in the loop.

There are many ways to close the connection either deliberately or accidentally – both of these cases are handled by the `try-except` block that encloses the chatting functions. If one side cancels the message dialog, the next `print` statement raises an exception that is caught by the `except` clause and the connection is terminated. If the connection breaks for any other reason, sending and receiving fails and both phones exit from the chatting loop.

The last lines in Example 58 are executed when the example starts. Here a dialog is shown which lets you choose whether you want to be the server or the client. Depending on the choice, either `chat_client()` or `chat_server()` is executed and the chatting may begin.

7.4 Phone-to-PC Communication

In this section, we describe how to connect your phone to your PC using Bluetooth RFCOMM (Figure 7.3). On the PC side, you can either use a Python script to communicate with the phone or any other application or programming language that can access the serial port. In Example 59, however, we rely on standard terminal emulator software, as described in Appendix B, which provides a convenient way to test the connection.

With the PC as a gateway or middleman, it is possible to control a wide range of applications and devices using a mobile phone, although the applications and devices do not have any native support for Bluetooth or mobile phones. Basically anything that can be controlled by a custom program on a PC, can be controlled by the mobile phone using the

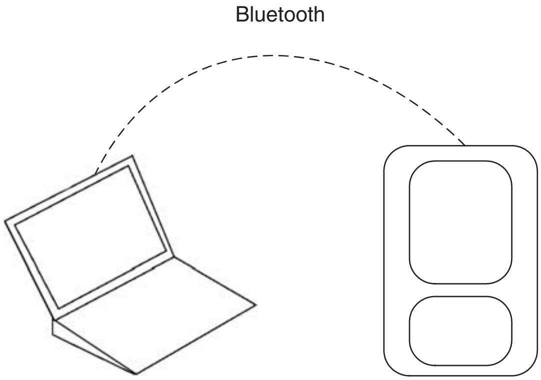


Figure 7.3 Bluetooth from phone to PC

Bluetooth link. This way, it is possible to build a PC-based personal video recorder, PowerPoint presentation or printer, which can be operated by a mobile phone interface.

Naturally, this requires that your PC supports Bluetooth communication, either internally or with a USB Bluetooth dongle. We also expect you to have installed Bluetooth drivers correctly; that is, Bluetooth must be fully working on the PC side.

To use Bluetooth for RFCOMM communication, some configuration is needed on the PC side. The process of setting up the RFCOMM serial port on Windows, Mac OS X, or Linux, is described in Appendix B. Once you can see the serial port service on your phone, you may proceed to the next section.

7.4.1 Communicating with the PC

The principles of RFCOMM communication with the PC are no different from the chat example, in which two phones were talking to each other. On the phone end, the code could look pretty much the same. However, here we consider only the option where the PC acts as the server and the phone connects to it. Technically, it is possible to use the phone as the server, but the Bluetooth configuration described in Appendix B holds only for the former case.

First, we make a small test client (Example 59) for the phone which can send and receive individual lines of text. This example can easily be used as a basis for any other type of communication: For instance, once you have learnt about JSON in Chapter 8, you can modify this client to transfer more complex data structures, such as lists and dictionaries, over the Bluetooth link. Also, your programs can use this method to transfer any files between the PC and the phone.

On the PC side, you can use any terminal emulator software, such as HyperTerminal or Screen, to communicate with the client. Appendix B

explains how to use one. If you have already used the PyS60 interpreter's Bluetooth console, you are already familiar with this setting.

Example 59: Bluetooth client

```
import appuifw, socket, e32

ECHO = True

def choose_service(services):
    names = []
    channels = []
    for name, channel in services.items():
        names.append(name)
        channels.append(channel)
    index = appuifw.popup_menu(names, u"Choose service")
    return channels[index]

def read_and_echo(fd):
    buf = r = ""
    while r != "\n" and r != "\r":
        r = fd.read(1)
        if ECHO: fd.write(r)
        buf += r
    if ECHO: fd.write("\n")
    return buf

address, services = socket.bt_discover()
channel = choose_service(services)
conn = socket.socket(socket.AF_BT, socket.SOCK_STREAM)
conn.connect((address, channel))
to_peer = conn.makefile("rw", 0)

while True:
    msg = appuifw.query(u"Send a message", "text")
    if msg:
        print >> to_peer, msg + "\r"
        print "Sending: " + msg
        print "Waiting for reply..."
        reply = read_and_echo(to_peer).strip()
        appuifw.note(unicode(reply), "info")
        if reply.find("bye!") != -1:
            break
        else:
            break
to_peer.close()
conn.close()
print "bye!"
```

The program starts by discovering nearby Bluetooth devices. You should choose your PC from the presented list. Then, the function `choose_service()` is used to show a popup menu that lists the services that are available on the chosen device. In this list, you should see an entry such as 'Serial Port' or 'PyBook', if everything is set up correctly on the PC side. The function returns a channel number that corresponds to the chosen service.

The Bluetooth address of your computer and the channel corresponding to its RFCOMM port are then known and we can establish a connection to the PC. We convert the connected socket to a convenient file object with the `makefile()` function. Now we have an active Bluetooth link to the PC.

As in the chat example, we handle communication in an infinite loop. The client is the first to send a message – if you choose to cancel the 'Send a message' dialog, the program exits. Otherwise, we send the line you typed to PC with the following expression:

```
print >> to_peer
```

Note that we add an additional character, `"\r"` or carriage return, at the end of the line. This is needed by the terminal emulator software on the PC side to start a new line properly.

In the chat example we used the file object's standard `readline()` function to receive a line. That would be a working option here as well, if we ran a program of our own on the PC side, but in this case we are communicating with a terminal emulator and that needs some special treatment.

The function `read_and_echo()` is a custom implementation of the standard `readline()` function. Compared to the standard function, there are two differences: the line may end in either a carriage return character, `"\r"`, or a new line character, `"\n"`; and the function may echo each character back to the sender, if the constant `ECHO` is set `True`. One character is read at a time, using the `fd.read(1)` expression. A terminal emulator expects that the other end is responsible for showing the characters typed, thus, unless we echo the characters back to the PC, you can send messages to the phone but you cannot see the characters while you type.

The received line is shown in a popup dialog. If the line contains the string ''bye!'', the connection is shut down. Thus, you may close the connection by typing ''bye!'' on the terminal emulator software.

You can now try this example: follow the instructions in Section B.2 on how to start a terminal emulator software PC with an RFCOMM serial port. Instead of using the standard Bluetooth console on the phone side, use the example above. If everything works correctly, the first message that you typed on the phone should appear on the terminal emulator. You can continue sending messages back and forth until you press cancel on the phone or send ''bye!'' from the PC side.

7.4.2 Creating Bluetooth Servers with PySerial

Section 7.4.1 showed how to make a phone client that communicates with the PC. However, we used a standard terminal emulator on the PC

side, so the messages that were sent to the phone were typed in manually. The most flexible approach, however, is to make servers of your own. This way you can access all resources on your PC, process information, send results to the phone and receive commands from it over a Bluetooth link without any manual attendance. This section shows you how.

You can access the serial port on the PC side most easily with a custom Python module, PySerial, which is available at ***http://pyserial.sourceforge.net***. Install the module as instructed on the Web page. Naturally you need a Python interpreter installed on your PC as well. You can find the newest one at ***www.python.org***.

Example 60 creates a simple server program for your PC, using PySerial, that receives messages from the phone and responds to them automatically. The functionality is demonstrated by a simple 'Guess my number' game: the server picks a random number between 1 and 10 and the phone user has to guess it.

Example 60: PySerial script running on PC

```
import serial, sys, random

if sys.platform.find("win") != -1:
    PORT = 0
elif sys.platform.find("linux") != -1:
    PORT = "/dev/rfcomm0"
elif sys.platform.find("darwin") != -1:
    PORT = "/dev/tty.pybook"

num = random.randint(1, 10)

serial = serial.Serial(PORT)
print "Waiting for message..."
while True:
    msg = serial.readline().strip()
    guess = int(msg)
    print "Guess: %d" % guess
    if guess > num:
        print >> serial, "My number is smaller"
    elif guess < num:
        print >> serial, "My number is larger"
    else:
        print >> serial, "Correct! bye!"
    break
```

You can use the client program in Example 59 to play the game. However, one minor modification is needed: this program is not interested in receiving back the characters it sent, so we can disable echoing in the client. Replace the line `ECHO = True` with `ECHO = False` in the client code.

In the beginning, we choose the port for communication based on the platform where the server is running. If you are using Windows, the line `PORT = 0` corresponds to the COM port that you set up for Bluetooth

communication in Section B.1. Note that indexing starts from zero, so `PORT = 0` corresponds to COM1 and so on. For Linux and Mac, the file name should be correct unless you modified the default name used in the instructions, in which case you should change the file name here accordingly.

The server is a straightforward one: once the serial port is opened and assigned to the variable `serial`, we can start communicating with the phone through it. The `Serial` object works like a standard file object, so we can use the familiar `readline()` function to read a line at time from the client. Each line should contain one number, which is then compared to the correct one, `num`, after which an appropriate hint is sent back to the phone. When extending the example, note that the PySerial module contains many useful options, such as timeout values, which can be used to tune the server. See the PySerial documentation for a full list of features.

To use this program, follow the instructions in Section B.2 but now instead of opening a terminal emulator, such as Screen or HyperTerminal, you need to execute Example 60 on the PC side. On the PC, a Python script is executed by typing the command `python btserver.py` into the command shell. The file `btserver.py` should contain the code for Example 60.

Even though the 'Guess my number' game might not seem fascinating in itself, you should notice how easily one can build software on the PC side that communicates with the phone. There are thousands of custom modules freely available for Python, which let you control and access various devices and applications on the PC. With this method, you can control any of those modules with your mobile phone over Bluetooth.

For example, you could turn your PC into a multiplayer game console which uses mobile phones as controllers – have a look at ***www.pygame.org*** to get started with making games in Python. If you are an artist, you can make stunning interactive installations using ***www.nodebox.net***, controlled by a mobile phone.

Section 7.4.3 shows how to control *any application* on your Mac OS X with your mobile phone. Even if you use Windows or Linux, you might want to have a look at the next section, since the basic idea can be applied with other operating systems as well.

7.4.3 Controlling Applications with AppleScript

The Mac OS X operating system includes a programming language called AppleScript. Its main purpose is to provide a simple mechanism to automate usage of any application. You can find more information about it at ***www.apple.com/applescript***.

Similar mechanisms also exist for other environments. For example, in Linux you can easily use Example 61 to execute any command or shell

script on the command line. Major desktop environments for Linux, KDE and Gnome, also support some AppleScript-like mechanisms to control applications programmatically.

Example 61 lets you, or possibly someone else, use your computer remotely. You should understand that doing this is potentially dangerous, especially if the program is executed in a public space. You should never run a server that lets the phone freely choose a command to execute. Instead, have a predefined list of commands that the phone is allowed to use and make sure that these commands cannot do any harm to your computer.

We need a special Python server program that runs on the Mac and executes an AppleScript file, as requested by the phone. Basically, the server is similar to Example 60 but, because of its specific purpose, this one contains even fewer lines of code.

Example 61: AppleScript interface running on Mac

```
import serial, os

ALLOWED_SCRIPTS = ["edit"]

ser = serial.Serial('/dev/tty.pybook')

print "Waiting for message..."
while True:
    msg = serial.readline().strip()
    if msg == "exit":
        print >> serial, "bye!"
        break
    elif msg in ALLOWED_SCRIPTS:
        print "Running script: " + msg
        os.system("osascript %s.script" % msg)
        print >> serial, "Script ok!"
```

The idea is that the phone client is used to choose one of the scripts to execute on the PC side. Once again, we can use Example 59 as the phone client. Again, echoing should be disabled by setting `ECHO = False` in the client.

In Python, the `os.system()` function is used to execute an external command. In this case, we execute the command `osascript` that is used to interpret AppleScripts on Mac OS X. However, the script name must exist on the `ALLOWED_SCRIPTS` list before it can be executed. This makes sure that a malicious user cannot connect to the server and execute any command she likes, for instance to format the hard disk. If the phone sends the line `exit`, the connection is shut down. Otherwise the user can execute as many commands as she wishes.

Naturally, we need some AppleScripts to be executed. Without going into details, we give here a simple AppleScript example that first activates the Finder application on the Mac and then opens a new file in the

TextEdit application. Make sure you have no other TextEdit file open when you run it. Write the following code to a file called `edit.script` and save it to the same folder as the above server program.

```
tell application "Finder"
activate
open application file "TextEdit.app" of folder\
       "Applications" of startup disk
end tell
```

The server is executed similarly to the previous server example. Now, however, the only sensible message to send from the phone is 'edit', which corresponds to the AppleScript file name without the ending '.script'. Once the message is received by the PC, you should see the TextEdit application start up.

Based on this teaser, you can start developing AppleScripts of your own that may control various applications, such as iTunes, iPhoto, QuickTime, DVD Player, Keynote, iSync or iCal. Just save the new AppleScript to some file that ends with '.script' and add its name to the `ALLOWED_SCRIPTS` list. For instance, by combining keyboard events from Chapter 5 to this example, you could control, say a DVD player, using the arrow keys on your phone.

7.5 Communication with GPS and Other Devices

Besides mobile phones and PCs, there are many devices that can communicate over Bluetooth. Many of them support RFCOMM, so you can directly apply the code in this chapter to a large number of use cases.

7.5.1 Connecting to GPS over Bluetooth

In this section, we connect to an external Bluetooth GPS reader to acquire GPS data (see Figure 7.4).

Some modern mobile phones, such as Nokia N95, come with an integrated GPS receiver. You can use PyS60 to obtain information from the internal GPS, as described in Section 6.4. Unfortunately, accessing the internal GPS requires some capabilities that are not available for the self-signed PyS60 interpreter (see Appendix A for more information). In contrast, you can use an external GPS receiver without any special capabilities just by connecting to it over Bluetooth.

Data that comes from the external GPS receiver follows a standard specified by the National Marine Electronics Association (NMEA). You can find a description of this format in many sources on the web, for instance, at ***www.gpsinformation.org/dale/nmea.htm***.

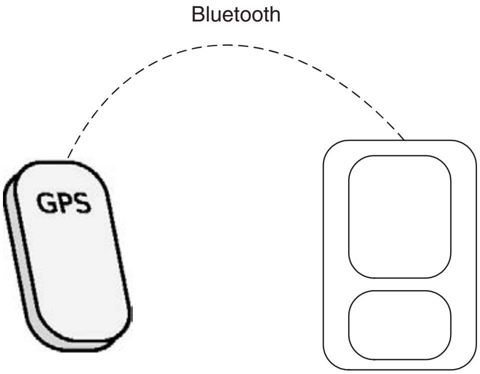


Figure 7.4 Bluetooth from phone to GPS

The format is line-oriented: the GPS sends one message per line. The line contains typically many fields, which are separated by commas. The line begins with the dollar sign, `$`, which is followed by a word that specifies the data type. The messages look like this:

```
$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47
```

This line has the data type `GPGGA` and it contains 15 fields. Most importantly, the third to sixth fields contain the latitude and longitude of the current position. On the above line, the latitude is 48.07.038′N and the longitude is 11.31.000′E. You can find this information and the meaning of other fields, in the NMEA specification. These coordinates can be readily plotted to many modern applications, such as Google Maps.

Besides the `GPGGA` lines, the data stream from the GPS receiver contains many other types of lines as well, as specified by NMEA. However, simple GPS-tracking software, such as the one presented below, might need only the `GPGGA` lines that specify the current position.

Example 62: GPS reader

```
import socket

address, services = socket.bt_discover()
print "Discovered: %s, %s" % (address, services)
target = (address, services.values()[0])

conn = socket.socket(socket.AF_BT, socket.SOCK_STREAM)
conn.connect(target)
to_gps = conn.makefile("r", 0)

while True:
    msg = to_gps.readline()
```

```
    if msg.startswith("$GPGGA"):
        gps_data = msg.split(",")
        lat = gps_data[2]
        lon = gps_data[4]
        break

to_gps.close()
conn.close()
print "You are now at latitude %s and longitude %s" % (lat, lon)
```

This example is based on techniques that are already familiar to you from the previous examples. First, you choose the GPS receiver from the list of discovered devices. It is not necessary to choose a service, as the GPS receiver provides only one. The socket is connected to the receiver in the usual way. If you know the address of your GPS receiver beforehand, or you have saved it to a file, you can skip the time-consuming `socket.bt_discover()` function and connect to the specified address directly.

We receive and skip lines from the GPS until we get the one that contains the current coordinates, that is, the one that begins with the string `"$GPGGA"`. After this, we exit from the loop and print out the coordinates.

You could extend the example, for instance, by saving the coordinates in a file so that you can plot them to Google Earth, or, you could receive the map of this location straight away with the MopyMaps! application, which is described in Section 9.2.

7.5.2 Connecting to Other Devices over Bluetooth

In Chapter 11, we present more real-world uses of Bluetooth. We show how to use Bluetooth to communicate with a sensor board kit, Arduino (see Figure 7.5), which is an extremely simple, yet powerful platform for building custom hardware. We also describe how PyS60 and Bluetooth were used to build an autonomous robotic vacuum cleaner and how to control sound-generation software called Max/MSP.

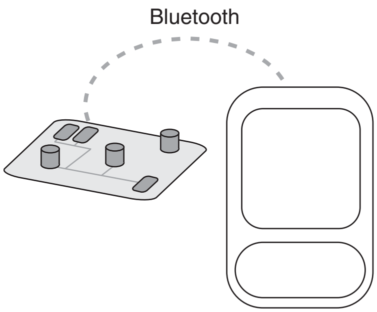


Figure 7.5 Bluetooth from a phone to a sensor board

As you will notice, these examples are similar to the previous examples that use RFCOMM. After reading these sections you have already seen everything needed to build practical, as well as artistic and esoteric, applications using Bluetooth!

7.6 Telephone Functionality and Contacts

The `telephone` module provides an API to the telephone functionality of the mobile phone, so it is possible to use a mobile phone as a phone! Compared to many other modules in this book, the `telephone` module is particularly simple.

The module includes only two functions: `telephone.dial()` and `telephone.hang_up()`. The `dial()` function accepts a telephone number as a string, as it may contain special characters, such as the plus sign. Note that if there is an call in process when the `dial()` function is called, the previous call is put on hold and a new call is established.

Example 63: Telephone

```
import telephone, e32

telephone.dial('+1412345602')
e32.ao_sleep(10)
telephone.hang_up()
```

If you start recording sound when a call is active, the phone call is recorded. See Example 26 for a simple sound recorder that can easily be combined with the `telephone` module. Ethical implications are left as an exercise for the reader.

The `contacts` module offers an API to the phone's address book. Although in the simplest form the address book could be a dictionary, which has names as keys and phone numbers as values, the modern mobile phone provides a sophisticated database which can handle rich forms of contact information. Correspondingly, the `contacts` module includes an extensive set of functions and objects that hook into the underlying contacts service that is provided by the operating system.

The PyS60 API documentation contains a detailed description of the module, so we do not repeat it here. Instead, we give an overview of the most typical use case: finding the phone number of an individual, given a name as a string. The following steps are needed:

1. Open the contacts database with `db = contacts.open()`.
2. Find an entry in the database which contains the desired name using `db.find(name)`. The matching entries, if any, are returned in a list. Each item of the list is a `Contact` object. Let's denote one such item with `match`.

3. The person's name can be found with `match.title`. In addition, a `Contact` object may have over 20 fields, such as name, job title, state and email address. If you need to call the person, typically the `mobile_number` field might contain the needed information. It can be retrieved with `field = match.find("mobile_number")`. This returns a list of fields, as the person may have multiple mobile phones. It is also possible that the requested field might not exist, in which case `None` is returned.
4. The `field` variable is an object of type `ContactField`. Most importantly, the actual value of the field can be found in `field.value`. This is, finally, the phone number string.

The following example is a straightforward application of the steps outlined above. It lets you search for a string, `name`, in the contacts database. It shows a selection list that contains the matching items and retrieves the mobile phone number for the chosen entry. Finally, it calls this number using `telephone.dial()`.

Example 64: Contacts

```
import contacts, appuifw, telephone
name = appuifw.query(u'Call to', 'text')
db = contacts.open()
entries = db.find(name)
names = []
for item in entries:
    names.append(item.title)
if names:
    index = appuifw.selection_list(names, search_field=0)
    num = entries[index].find('mobile_number')
    if num:
        telephone.dial(num[0].value)
    else:
        appuifw.note(u'Missing mobile phone number', 'error')
else:
    appuifw.note(u'No matches','error')
```

Similar functionality was used in the Hangman SMS server in Section 4.5.

7.7 System Information

The `sysinfo` module offers an API for checking the system information of the mobile device. The following functions are provided:

- `battery()` returns the current battery level ranging from 0 to 7. If using an emulator, value 0 is always returned.
- `imei()` returns the IMEI of the device as a Unicode string. If using an emulator, the hardcoded string `u"000000000000000"` is returned.

- `active_profile()` returns the current active profile as a string, which can be one of the following: 'general', 'silent', 'meeting', 'outdoor', 'pager', 'offline', 'drive', or 'user <profile value>'.
- `display_pixels()` returns the width and height of the display in pixels.
- `display_twips()` returns the width and height of the display in twips (screen-independent units to ensure that the proportion of screen elements are the same on all display systems). A twip is defined as 1/1440 of an inch, or 1/567 of a centimeter.
- `free_drivespace()` returns the amount of free space left on the drives in bytes.
- `max_ramdrive_size()` returns the maximum size of the RAM drive on the device.
- `total_ram()` returns the amount of RAM memory on the device.
- `free_ram()` returns the amount of free RAM memory available on the device.
- `total_rom()` returns the amount of read-only ROM memory on the device.
- `ring_type()` returns the current ringing type as a string, which can be one of the following: 'normal', 'ascending', 'ring_once', 'beep', or 'silent'.
- `os_version()` returns the operating system version number of the device.
- `signal_bars()` returns the current network signal strength ranging from 0 to 7, with 0 meaning no signal and 7 meaning a strong signal. If using an emulator, value 0 is always returned.
- `dbm()` returns the current network signal strength in dBm. If using an emulator the returned value is always 0.
- `sw_version()` returns the software version as a Unicode string. If using an emulator, it returns the hardcoded string `u"emulator"`.

Example 65: Sysinfo

```
import sysinfo
print "Battery level: %d" % sysinfo.battery()
```

7.8 Summary

In this chapter we have covered the Bluetooth functionality of the phone. We shared photos with a nearby phone and made a chat application for a pair of phones. We showed how simple it is to communicate with a PC, either with a standard terminal emulator or server software of your own.

Traditionally, people have connected their mobile phones to PCs to synchronize contact information or calendars – considering the plethora of possibilities, this just scratches the surface. The mobile phone can take full control of the PC using simple methods, which was shown by the AppleScript example.

We also demonstrated how to use the telephone programmatically and how to find people in the address book. Even though telephone is not the most trendy communication method nowadays, it is undoubtedly the most ubiquitous and most compatible, which makes it still highly relevant. In Section 7.7, we gave an overview of the `sysinfo` module which is a treasure trove of information about the system state.

Now that you can fully control everything in the five meters around you, or at least everything that understands Bluetooth, it is time to look further. The next two chapters introduce you to networking, which gives you the necessary tools to communicate with practically any server in the world.

You should keep in mind that, even on the Internet, your greatest asset may be the five meters around you: when anyone on the Internet needs information from that area, you and your mobile phone may be the only gateways that can provide the desired information on the spot.

8

Mobile Networking

The modern mobile phone is a personal computer whose primary purpose is to keep its user connected with the outside world. This is where PyS60 becomes most fun: you can innovate and experiment with the most insane, amazing and productive combinations of you and others, our physical environment and all the digital information in the world.

After these grandiose motivational words, we can crawl back to the technical details. Writing distributed applications, including the ones discussed in this chapter, are more difficult to get to work right than programs that do not communicate. The reason is simple: you have to keep three independent components synchronized instead of a single program.

The client, the server and the network between them are all running independent, complex pieces of software that are only kept together by the force of protocols. In addition, with mobile phones, one of the components keeps changing all the time. Depending on your location, the network may be non-existent or anything between a low-bandwidth GSM connection to a broadband wireless LAN. Fortunately for us, a dynamic programming language, such as Python, makes it possible to adapt to changing environments with relative ease.

Even better news is that the most typical networking tasks are made really simple in PyS60. You can download anything from the web with one line of code and you can upload anything to your own website almost as easily. These basic tasks are introduced first in Section 8.1.

If you are primarily interested in interacting with various web services, you do not have to bother about the underlying techniques too much. You can read only Section 8.1 and then proceed directly to Chapter 9, which deals with cool web-based applications without being heavy on technical details. Note that, in this case, you still need to install the JSON extension module as instructed in Section 8.2.2.

We give a guide on how to set up a development environment for mobile networking in Section 8.2. Then a toolbox of protocols is

introduced in Section 8.3, which forms the basis for networking. A brief introduction to server-side code is given in Section 8.4, which helps you to get started on that side as well.

Finally, Sections 8.5, 8.6 and 8.7 describe three patterns of advanced networking for mobile phones. Each pattern is explained using a fully working and non-trivial example, which can be used as a basis for further prototyping.

Examples in this chapter are stand-alone and platform-agnostic: you can use them as they are or you can easily adapt them to work with your platform of choice, whether it be Django, Ruby on Rails, .NET or Erlang.

8.1 Simple Web Tasks

8.1.1 Downloading Data from the Web

You can download any HTML page, image, sound or any other file from the web in one line of code.

Example 66: Web downloader

```
import urllib
page = urllib.urlopen("http://www.python.org").read()
print page
```

The script prints out the HTML contents of the Python home page. The module `urllib` contains functions related to web communication. Whenever you need to download something from the web, you can use the `urllib.urlopen()` function, which returns the file contents to your program so that they can be further processed.

On the other hand, if you want to download a file from the web and save it locally, you can use the `urllib.urlretrieve()` function to save the file directly to a given file name, as shown in Example 67. If the URL contains unusual characters, such as spaces, you can use the `urllib.quote()` function to encode them properly.

This example downloads the Python logo from the Python website and shows it using the phone's default image viewer.

Example 67: Web file viewer

```
import urllib, appuifw, e32

URL = "http://www.python.org/images/python-logo.gif"

dest_file = u"E:\\Images\\python-logo.gif"
urllib.urlretrieve(URL, dest_file)
```

```
lock = e32.Ao_lock()
viewer = appuifw.Content_handler(lock.signal)
viewer.open(dest_file)
lock.wait()
```

This example uses the function `urllib.urlretrieve()` to download data from a URL to a local file. When the data has been saved, the default viewer associated with the URL's file type is used to open the file. You can change the URL to point at JPEG images, web pages, MP3 files or any other file type supported by your phone.

The `Ao_lock` object is needed to keep the script alive until the viewer is finished. You can easily extend this example into a generic web downloader that asks the user for a URL to download and then shows the file using the default viewer. You can implement it as an exercise.

8.1.2 Uploading Data to the Web

Sometimes you want to upload photos, sounds or other files from your phone to your website. Example 68 presents a simple PyS60 script that takes a photo and sends it to a website. On the server side, we show how to receive the photo in PHP. You can easily adapt the example to the web back end of your choice.

Example 68: Photo uploader

```
import camera, httplib

PHOTO = u"e:\\Images\\photo_upload.jpg"

def photo():
    photo = camera.take_photo()
    photo.save(PHOTO)

def upload():
    image = file(PHOTO).read()
    conn = httplib.HTTPConnection("www.myserver.com")
    conn.request("POST", "/upload.php", image)
    conn.close()

photo()
print "photo taken"
upload()
print "uploading done"
```

The function `photo()` takes a photo and saves it to a file. Then, the function `upload()` reads the photo (JPEG data) from the file and uses HTTP POST to send it to the server. Here we use the module `httplib` instead of `urllib`. This module gives a lower-level access to HTTP and makes fewer assumptions about the data sent.

On the server side, you can use whatever method you want to receive the data. Here, we give a minimal example in PHP that receives the photo and saves it to a file:

```
<?php
$chunk = file_get_contents('php://input');
$handle = fopen('images/photo.jpg', 'wb');
fputs($handle, $chunk, strlen($chunk));
fclose($handle);
?>
```

There are many possible ways to send files to a server, although this might be the simplest one. In Section 8.4, we show how to build a custom server to receive photos from the phone. In Section 9.4, we build a fully working application, InstaFlickr, that uploads photos to Flickr using a slightly different technique.

8.2 Setting up the Development Environment

Before you start writing any networked programs, you should make sure that you have a working network. Since development requires lots of trial and error, you should be able to update code on your server, as well as on your phone, without too many steps.

In the following sections, we show how to set up a test environment in four different settings:

- Environment A: a WiFi-enabled phone, a local WiFi network and a PC connected to it; the PC works as a test server and no Internet access is needed.
- Environment B: a phone with Internet access through a GSM or 3G data plan; the PC works as a test server and you can connect to it from the Internet, possibly through a firewall.
- Environment C: a phone with Internet access, either through a GSM or 3G data plan or a WiFi network; you need shell (SSH) access to an external test server.
- Environment D: a phone with Internet access, either through a GSM or 3G data plan or a WiFi network; you need access to an external web server which is used for testing.

If you have already developed server-side code for other purposes, you can probably use your existing environment for mobile development as well. If you do not have previous experience in this area, environment A is likely to be the easiest option to start with.

8.2.1 Preliminaries

It is useful to know what we are aiming to achieve. Here is a quick reminder of how the Internet works. Each computer or host has an *IP address*, which is of the form 123.45.54.56. Data is transferred between hosts in *packets*. Since one host can run multiple networked programs at the same time, each program is assigned a *port* on which it can receive packets. Each packet contains the IP address and port of its destination, so the packet can find its way to the correct program at the right host.

Usually, a TCP/IP connection is opened between two hosts, which ensures that the two hosts can communicate reliably with each other. The TCP protocol is built on top of the IP protocol. In particular, TCP connections are established using IP addresses. You can think of the connection as a pipe through which the packets are transferred. It is the job of a firewall to control who can open connections to which ports on a host. These connections are the core of Internetworking. Everything else is built on top of this, including the web.

The TCP connection is similar to the Bluetooth connection established by the RFCOMM protocol, which we used in Chapter 7. Here, however, an IP address is used instead of a Bluetooth address and a port instead of a channel.

Our goal is to get your mobile phone, which has an IP address, to open a connection to your server, which operates behind another IP address. This should be straightforward once you know both addresses. Unfortunately, your server may be behind a firewall or a Network Address Translation (NAT) device which hides your computer behind an IP address of its own. These devices may prevent the connection being opened. It is possible to evade these obstacles but it requires further tweaking.

8.2.2 Install JSON Module

To use the JavaScript Object Notation (JSON) protocol, which is used in many examples in this chapter and in Chapter 9, you have to install a JSON extension module to PyS60. This is required since JSON is not included in the standard PyS60 library, unlike the other modules we have used this far.

The module is installed as follows:

1. Go to the book website at ***www.mobilepythonbook.com***.
2. There, conveniently on the first page, you will find a link to the JSON module.
3. Download the file, `json.py`.

Extension modules must be installed to the `E:\Python\Lib` directory in PyS60. If this directory does not exist already, create it in a similar way to the `E:\Python\` folder, as instructed in Chapter 2.

Upload `json.py` to `E:\Python\Lib` on your phone. You can use your usual way to upload the file. If you use an S60 2nd Edition device, choose 'Install as Python lib module'.

You can test that the module is installed correctly with the following script:

```
import json
print "JSON OK!"
```

If the 'JSON OK!' line is printed out, the module is installed correctly. The JSON module, `json.py`, will be needed by server-side examples as well. You need to copy this file to the folder on your PC where you run the examples.

8.2.3 Networking Environments

Figure 8.1 summarizes the networking environment for the modern mobile phone. Typically, your phone communicates with a GSM (or 3G) base station. The base station routes traffic, by way of your operator's data center, to the Internet. This way you can, for example, read web

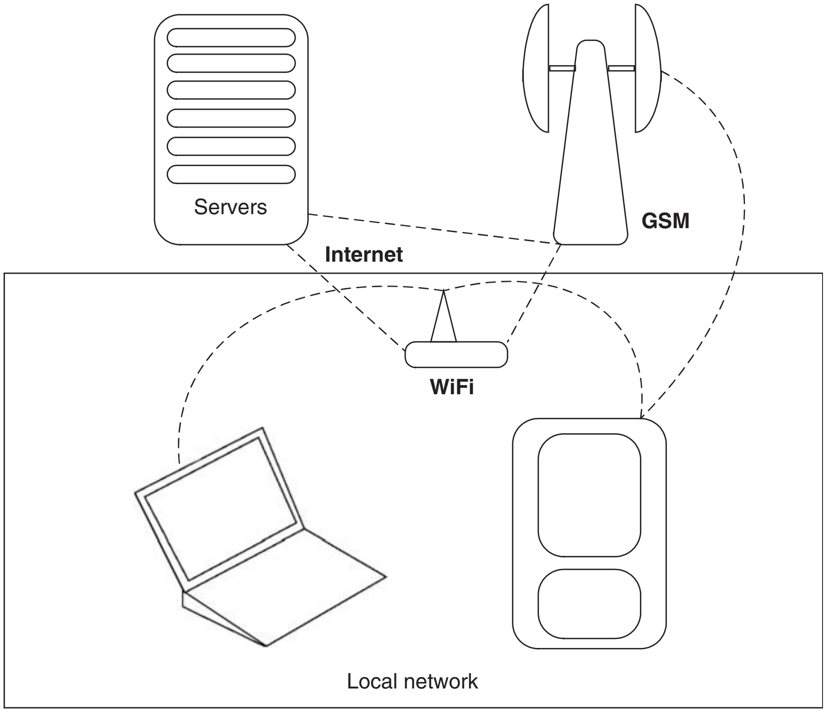


Figure 8.1 Networking environment of a modern mobile phone

pages on your phone – assuming that your phone service includes a data plan.

Depending on your data plan, your operator may either charge you for the amount of traffic or have a flat fee. Many examples in this chapter generate less than a kilobyte of traffic, which is almost nothing nowadays; it is less than a typical web page. As long as you do not transfer sounds, images or video, it is hard to generate even a megabyte of traffic, unless your script generates data in an infinite loop.

In the following, we give details of four network environments that are suitable for rapid development. If your phone has a WiFi capability, you should probably use environment A, as it is flexible and does not cost anything to use. If you plan to use GPRS or 3G instead, environment C is a suitable choice, given that you have access to a server already – if not, then environment B is a feasible option. Environment D is easy to set up but you will have to adapt the examples to your particular environment.

Environment A: Local wireless network

If your phone has a WiFi capability, it can join a local wireless network, assuming you have such a network set up. In the best case, you also have a laptop or other computer which can join the same local network and which you can use as a test server. In this case, you do not have to pay for the traffic and you have full access to the server. You can probably avoid many problems with firewalls too, when you operate in a local network.

Environment B: Phone Internet access and a PC server

If your phone is not WiFi-capable, your test server has to be visible to the Internet as the traffic goes by your operator's network. In this case, you have two options: use your local PC as a server or use a rented or co-located server.

It is not always easy to use a PC as a publicly accessible server. You may face problems with local firewalls and NAT devices. You should allow inbound traffic in your firewall to a selected port, say 9000. If you use NAT, you should enable forwarding from port 9000 on your NAT device to port 9000 on your local computer.

In Section 8.2.4, you will see how to find out your local IP address. Your router's or firewall's documentation should help you here also. However, be careful with your firewall configuration; if you leave it disabled by accident, your computer becomes easy prey for bots and crackers.

Environment C: Phone Internet access and an external test server

Renting a virtual server is inexpensive nowadays. For mobile development, you need a server with a shell account, Python, a public IP address

and some ports open in the firewall. This is also a typical setting in a company or university which often has several servers accessible from the Internet.

Environment D: Phone Internet access and an external web server

If you are only interested in developing web-based services, a web hotel which supports CGI, Ruby on Rails or a web back end of your choice is sufficient. Note, however, that you cannot use the example servers in this chapter without adapting them to use your web back end.

8.2.4 Testing the Connection

In this section, we test that your network environment is correctly set up for mobile development. According to your network environment, follow instructions in one of the sections below. Except in environment D, we first have to find out the server's IP address and then test the connection using a simple test server.

Environments A, B and C require a working Python installation on your server. You can find Python for Windows, Linux and Mac OS X at the Python home page at ***www.python.org***.

Environment A: Local wireless network

First you have to find out your IP address in your local network.

In Linux and Mac OS X you can use the following Python script to find out all IP addresses that have been assigned to your computer. You may have several of them, for instance, if you have both wired and wireless network adapters on your laptop. Make sure that you have joined your local wireless network before executing the following script:

```
import os, re
txt = os.popen("/sbin/ifconfig").read()
for t, ip in re.findall("inet (addr:)?([\d\.]+)", txt):
    print ip
```

Save the code above to the file `findip.py` and execute it on the command line using the command `python findip.py`. Do not worry about the code of this script.

In Windows, you have to execute command `ipconfig` in your command shell to find out the addresses.

If you see an address starting with '192.', it is probably your local IP address. Special address '127.0.0.1' is used only for internal communication by your computer and it is not the one we are interested in. If you cannot immediately recognize your local address, you can try them all. One of the addresses is the address of your test server.

Environment B: Phone Internet access and a PC server

You have to find out the server's external IP address. Open a web browser on your server (PC) and go to one of the following websites:

- ***http://checkip.dyndns.org***
- ***www.ip-address.com***
- ***http://whatismyipaddress.com***

Write down the IP address that the site reports. This is the address of your test server.

Environment C: Phone Internet access and an external test server

You have probably seen the address of your server already. If you can log in to the server using Secure SHell (SSH), you can use the address that is used by SSH for mobile development.

Environment D: Phone Internet access and an external web server

You probably have a domain name, such as www.mydomain.com, assigned to your server. The service's documentation should tell you how to update web pages at that address. You should be able to make a simple HTML form containing one text input box and a simple script that handles it on the server side.

The script should return `"Hello \$name"` where `\$name` is the text typed in the form. There are plenty of tutorials on the web showing how to do this using the web back end of your choice. Once this works, you have a test environment in place and you can proceed to Section 8.3.

Testing the server in environments A, B and C

Next we create a simple server script. The script waits for a new connection to be initiated. It then shows the client's IP address and the first line in the request. You need to execute this script on your PC (in environments A and B) or on your external server (environment C). The script waits for incoming connections in an infinite loop. You can interrupt the server by pressing Ctrl-C.

Example 69: Test server

```
import SocketServer

class Server(SocketServer.TCPServer):
    allow_reuse_address = True
```

```
class Handler(SocketServer.StreamRequestHandler):
    def handle(self):
        print "CLIENT IP %s:%d" % self.client_address
        print "Message: " + self.rfile.readline()

server = Server(('', 9000), Handler)
print "WAITING FOR NEW CONNECTIONS.."
server.serve_forever()
```

Save the code above to the file `testserver.py` and execute it on the command line using the command `python testserver.py`. This server works in the same way in Linux, Mac OS X and Windows, given that you have Python installed on your PC.

This test server uses some Python concepts, such as custom objects, which have not been introduced earlier in this book. You can find more information about creating objects, for example in the Python tutorial. However, for purposes of this book, you do not have to care about the details. Since this book is not about making Python software for servers, you can just skim through the following description. We use similar examples on the server side later in this chapter, so basic understanding of this example is useful.

The script is based on Python's standard `SocketServer` object, which takes care of waiting for new incoming TCP connections. Once a connection has been established, `SocketServer` calls the given callback function, which then handles the actual request.

We modify one parameter, `allow_reuse_address`, in the standard server. This modification makes it possible to re-use the same server port again if the server is restarted. In the default case, a closed port has a quarantine period during which it cannot be re-used.

The callback function must belong to a `StreamRequestHandler` object. The function is given one special parameter, `self`, that is used to access the object's internal variables. `self.client_address` contains the client's IP address and port in a tuple. Data from the client is read from `self.rfile`, which acts like a file object, in a similar way to the `serial` object in the Bluetooth server in Example 60.

The server object is created with two parameters in a tuple, the server port (9000) and a reference to the `Handler` object. The server port should correspond to the port which you have opened in your firewall. Examples in this chapter use port 9000, but you can use some other port. Note that port numbers below 1024 are reserved for registered services, so it is better to use a port number between 1025 and 65535.

If you have a local firewall, you should allow all incoming connections to the port which you used above (for example, 9000). If you use environment B and you have an Internet router with NAT, you have to enable port forwarding from the router to your PC, from and to the

port you specified above. Port forwarding may require you to specify the local IP address of your PC. Follow the instructions for environment A to find it.

Once your server is up and running, that is, it says 'WAITING FOR NEW CONNECTIONS', you can try to connect to it. We test the connection with a normal web browser, first on the PC and then on your mobile phone. If it works with the browser on your phone, it will work with PyS60, too.

First, open a web browser on your PC. Type in your server's address, which you found above, and specify the correct port. For example, a local address might look like `http://192.168.0.2:9000/`. When your browser tries to connect to the server, the server calls the request handler function `handle()`. The request handler then prints out the client's IP address and the first line of the HTTP request that the browser sent to the server.

If you see some lines printed out by your server script, the connection works correctly. The browser shows just an empty page, or it may show an error, as the server does not respond to the request. If the server script does not print out anything, a firewall might be blocking the connection or you may be trying to connect to the wrong IP address. If you found several possible IP addresses in environment A, try each of them until a connection is established.

After you have managed to establish a connection from your web browser on the PC to the server, you can try the same with your phone. Open the web browser on your phone and type in the server's IP address and port, exactly as on the PC.

If the connection is established correctly, you should see some lines printed out by the server script. If the connection works on your PC but nothing happens when you try to connect with your phone, there might be something wrong with the phone's network settings in general.

To find this out, try to connect to any well-known website, such as ***www.google.com***, with the phone's browser. If this succeeds, you have a working Internet connection and the culprit is probably a local firewall. If you cannot connect to any site with your phone, you should set up the Internet connection properly on the phone first, following the instructions given by your operator or service provider.

If you can now connect from your phone to your server, you have a working test environment. If you cannot, don't feel desperate. After all, the culprit might be a firewall or a NAT which is just working as it should.

You can find many troubleshooting guides on the web if you search for phrases such as 'port forwarding tutorial' or 'firewall tutorial'. Note that, in environments A and B, the IP address may change when your PC or router reboots. If your environment suddenly stops working, repeat the steps above to find out the new address.

8.3 Communication Protocols

A communication protocol defines how to move data from place A to place B. Depending on the application, one has to balance several different factors, such as latency, addressing and data encoding, when choosing the best protocols for the task – no single protocol is suitable for all applications. The situation is similar to the transportation of physical goods. You have to choose the proper packaging, routing and mode of transportation (air, ground or sea) for your cargo.

Luckily, some common protocols are enough for almost all applications. This chapter focuses on a stack of three protocols, depicted in Figure 8.2.

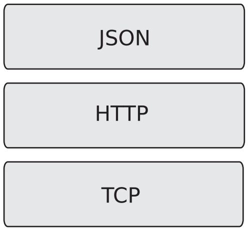


Figure 8.2 Protocol stack used in this book

It is likely that these protocols will serve all your needs for communication. Most of the Internet works on TCP and the web is built on top of it using HTTP. For this reason, we use the terms 'web server' and 'HTTP server' interchangeably. The topmost protocol, JSON, is a recent addition to the stack and it is typically used by modern web applications.

The protocols are complementary to each other: TCP is responsible for establishing connections between two IP addresses and moving packets between the two endpoints. Nowadays, HTTP is used as a common way to access services (including web pages) behind URL addresses. It uses TCP to transfer data between the client and an HTTP server. JSON is used to encode data structures, such as lists and dictionaries, in a string that can be sent easily over HTTP.

Imagine that you want to move a list of integers, for instance the numbers 1, 2, 3, from a PyS60 program to a web service. This is depicted in Figure 8.3, where the dark line represents the steps for sending and receiving the list; information on the highlighted rows is handled by the protocol written in the corresponding position on the dark line. You start by encoding the list to a string using JSON. You pack the string in an HTTP request which is targeted at a specific URL. After this, the HTTP client library uses TCP to establish a connection to the server which hosts the URL and sends the HTTP request over the newly established connection to the server.

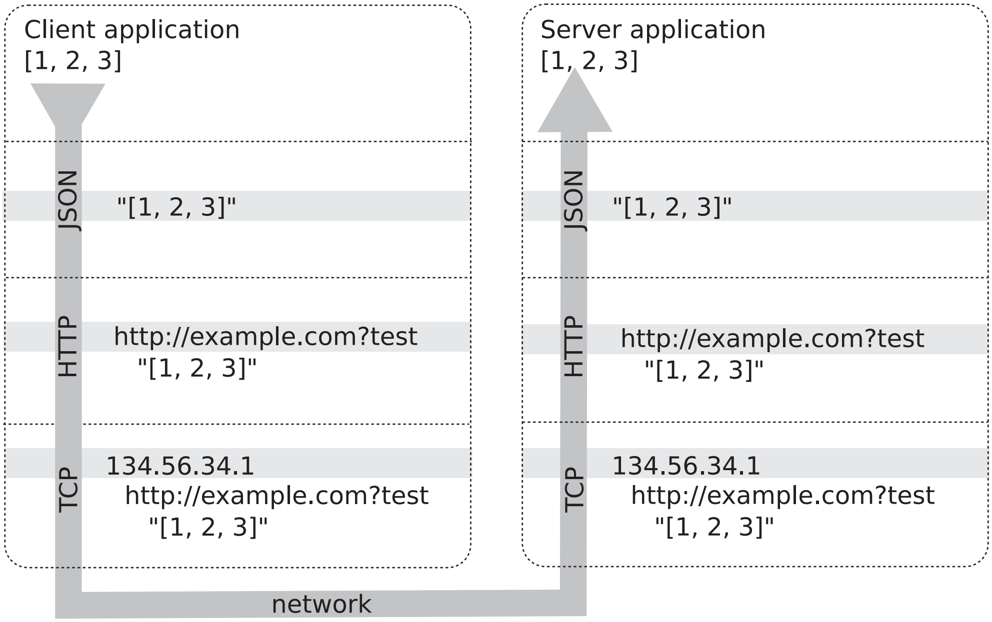


Figure 8.3 Sending data from a client to a server using the full JSON/HTTP/TCP protocol stack

The server unfolds the protocols one by one. First, it accepts the TCP connection, receives the HTTP request and finally it finds the JSON message in the request. By decoding the JSON message, the server-side application ends up with the original list.

This process required six steps in total – three on the client side and three on the server side. In PyS60, however, you can send a request and receive a response using this protocol stack in two lines of code. Take a sneak peek at Example 86 and see the function `json_request`.

In some cases, you do not need the full protocol stack. For example, if you only need to transfer plain strings (note that any files can be represented as strings or sequences of bytes), HTTP without JSON is enough. If you do not have a web server already in place, JSON-encoded strings over TCP, without the HTTP layer, might be a simple and working solution. A simple solution like this is often agile, flexible and robust by definition. There is no need to use the latest buzzword-compliant, enterprise-level, middleware framework, if all you need is a quick prototype for your newest killer application. In the following sections, you see how the protocols are used in practice.

8.3.1 TCP Client

We start with a plain TCP client. If you only have access to a web server, as in environment D above, you can skip this example and proceed to Section 8.3.2.

On the server side, we use the server in Example 69. It is almost the smallest sensible TCP server implementation in Python. Start the server script as instructed in Section 8.2.4.

The TCP client, which runs on your phone, is in Example 70.

Example 70: TCP client

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("192.168.0.2", 9000))
out = sock.makefile("rw", 0)
print >> out, "Hi! I'm the TCP client"
out.close()
print "Client ok"
```

The endpoints in a TCP connection are called sockets. Logically, the module `socket` includes the required Python interface. First, we create a new `socket` object: It is initialized with two parameters which specify the type of the protocol. The parameters used above specify a TCP connection. You might remember that the socket object was introduced in Chapter 7, where it was needed for RFCOMM communication.

The next line opens a connection to the server. You have to change the IP address to correspond to your server's real address – it is the same one which you used in Section 8.2 to test the connection. If you do not run the server script in port 9000, change the port number as well.

On the S60 platform, it is possible to define multiple networking profiles. For instance, you can have one profile for a wireless network at home and another for a GPRS connection. Whenever you want to open a network connection, you have to choose the profile through which the connection is established.

By default, a popup menu, like the one in Figure 8.4 is shown automatically by the function `socket.connect()`. If the dialog does not include a desired profile, it has to be configured in the phone's connectivity settings. In PyS60, it is possible to define the default access point, which prevents this dialog from opening (see Section 8.3.4).

Once the execution returns from the `connect()` function, a new connection has been established successfully. If `connect()` raises an exception, this is most often because of a wrong IP address or wrong port or a firewall blocking the connection. If this happens, you should check that your test environment works correctly by following the instructions in Section 8.2.

For convenience, we make the new connection look like a file object by calling the `makefile()` function. The parameter 'rw' specifies that we can both read from and write to the socket. The connection object



Figure 8.4 Access point selection dialog

`out` behaves exactly like its counterparts in the RFCOMM examples in Chapter 7. If everything works correctly, you should see 'Hi! I'm the TCP client' printed out on your server.

It is important to note that you cannot know how much data the other side has sent, or is going to send, through the TCP connection. To handle this situation, there are three alternatives:

- Read everything with the `read()` function without any parameters. Reading continues until the other side closes the connection.
- Read a fixed number of characters (bytes) at time, for instance, `read(10)` would read 10 bytes at most.
- Read a character at time until a character denoting the message end, such as the new line character, is received. The standard `readline()` function and the `read_and_echo()` function in Example 59 use this approach.

In the previous examples, we have used the `readline()` approach because of its simplicity. Typically a protocol that works on top of TCP, such as HTTP and JSON, will take care of this issue internally.

8.3.2 HTTP Client

The modern web is a diverse and complex environment. Even though it is possible, and actually quite easy, to load one web page from some pre-defined site using a plain TCP connection, writing a generic library which would work with any website would be a major effort.

However, as we saw in Section 8.1, PyS60 makes interacting with the web really smooth, thanks to the `urllib` module that implements an HTTP client in Python. That section presented many examples based on the `urllib` module and many more will follow in Chapter 9.

8.3.3 JSON Client

From Python's perspective, the TCP protocol only transfers strings between hosts. The HTTP protocol transfers strings as well, but it includes a simple method to encode key–value pairs, in practice a dictionary object, to a request string using the `urllib.urlencode()` function.

Technically speaking, this should suffice for all applications – assuming that every application finds a way to encode its internal data structures to strings and back. However, as there is only a small number of basic data structures, namely strings, numbers, Boolean values, lists and dictionaries, it makes sense to provide a common way to encode them. This saves you from the error-prone task of parsing data back and forth. JSON is a solution to this task.

JSON is mainly used as a lightweight data interchange format between client-side JavaScript code, which runs inside a web browser, and the web server. The protocol is textual and pleasantly human-readable. The encoding represents closely the way simple data structures are initialized in the JavaScript code and also in Python. The official home page for JSON is at ***http://json.org***.

Several factors make JSON a nice companion to PyS60:

- It is a lightweight, human-readable protocol.
- Encoding and decoding can be performed with a single function call.
- The data types it supports map well to Python.
- You can use it to communicate both with PyS60 and web clients, which simplifies server software.
- It is widely supported by many web frameworks, including Django, TurboGears and Ruby on Rails, and by many web services, including Yahoo! and Flickr, as you will see in Chapter 9.

The best way to understand JSON is to open the Python interpreter on your PC and see how different encodings look like, for instance as follows:

```
>> import json
>> json.write("hello world")
'"hello world"'
>> json.write({"fruits": ["apple", "orange"]})
'{"fruits": ["apple", "orange"]}'
>> json.write((1,2,3))
'[1,2,3]'
```

```
>> json.read(json.write([1,2,3]))
[1,2,3]
```

To try this on your PC, open the Python interpreter on the same directory where `json.py`, which you downloaded earlier from the book website, resides.

As you can see, JSON encodes the data structures to strings almost exactly as they are written in Python. Tuples are a notable exception as they are not recognized by JSON and they are encoded as lists instead. The last line shows that the string encoded by `json.write()` is decoded back to the original data structure by the function `json.read()`.

You can use JSON to save data structures in a file as well. As an exercise, you can replace our *ad hoc* format for saving dictionaries in Chapter 6 with JSON.

Next, we have a look at how to use JSON with HTTP. This example uses a web service by Yahoo! to retrieve information which is encoded in JSON. As this example relies on the Yahoo! API, which might have changed since the book's publication, you should first check that this service is still available and works correctly. The following URL should produce some output on your web browser:

http://developer.yahooapis.com/TimeService/V1/getTime?appid=MobilePython&output=json

If it does, you can try out Example 71 on your phone.

Example 71: Yahoo! web service test

```
import urllib, json, appuifw, time

URL = "http://developer.yahooapis.com/TimeService/V1/" +\
                 "getTime?appid=MobilePython&output=json"

output = json.read(urllib.urlopen(URL).read())
print "Yahoo response: ", output
tstamp = int(output["Result"]["Timestamp"])

appuifw.note(u"Yahoo says that time is %s" % time.ctime(tstamp))
```

The web service is uninteresting: it just returns the current time. Web services, such as this one, are accessed through normal `urllib` functions, in the same way as any other resource on the web. The requested URL specifies the service and its parameters.

In this case, the result is a string that includes a JSON-encoded dictionary with contents that are specified in the Yahoo! API documentation. If the example generates an error, it may be because of API changes on the Yahoo! side. You can check the current documentation provided by Yahoo! and try to fix the example. Chapter 9 will cover many more examples like this one.

8.3.4 Setting the Default Access Point

After trying out the previous examples, you have seen that PyS60 shows the access point selection menu, like the one in Figure 8.4, almost always when a new network connection is about to be opened.

On some phone models, the menu is shown only the first time when a network connection is opened on the PyS60 interpreter and the interpreter must be restarted to choose another access point.

PyS60 provides the necessary functions, as a part of the `socket` module, to set a desired access point programmatically. Example 72 shows the access point selection dialog and sets the default access point, so the dialog will not be shown during subsequent connection attempts.

Example 72: Set the default access point

```
import socket

ap_id = socket.select_access_point()
apo = socket.access_point(ap_id)
socket.set_default_access_point(apo)
```

The function `select_access_point()` pops up the dialog and returns the access point ID, `ap_id`, that is chosen by the user. The function `access_point()` converts the ID to an access point object. The returned object is then given to the function `set_default_access_point()`, which sets the default access point.

In Chapter 9, the EventFu and InstaFlickr applications show how these functions can be used in practice. The PyS60 API documentation includes some further examples of possible uses of these functions.

8.4 Server Software

The simple TCP server that was presented in Example 69 has a major drawback: it can serve only one client at a time. If only a few clients use the server, it is unlikely that several clients will try to connect to it simultaneously and each client will get a response without a noticeable delay. However, this approach does not scale.

Writing robust, secure and scalable server software from scratch is not easy. Fortunately, in most cases you do not have to worry about this. If you are experimenting with new ideas or you are making a prototype for a restricted number of people, you can freely use any approach that works, like the simple TCP server above. However, keep security in mind in these cases as well – a protected intranet or a correctly configured firewall keeps you safe with only little additional effort.

Remember that everything sent by TCP, HTTP or JSON is directly readable by anyone who can tap into the network you use. If you use an

open or weakly encrypted WiFi network, capturing the traffic is especially straightforward. Do not rely on passwords sent over these protocols and do not assume that requests always originate from a trusted client.

PyS60 supports Secure Socket Layer (SSL) and Secure HTTP (HTTPS). These protocols are a must if you need to transfer any sensitive information. More information about these protocols can be found in the PyS60 API documentation.

If you are building a production system or you are about to release your code to the public, it is usually a good idea to rely on an existing server framework, such as Twisted. An even easier approach is to make your server available as a web service, so you can rely on any web framework, such as Django or Ruby on Rails, for security, scalability and other advanced features.

8.4.1 JSON Server

In this section, we extend Example 69 to do something useful. Technically, Example 73 demonstrates how to use JSON over plain TCP. The example consists of a client program that runs on your phone and simple server software that runs on your server.

The phone client lets the user take named photos which are automatically sent to the server. The photo is taken immediately when the `name` dialog closes. You can easily add a viewfinder to this program, for instance, based on Example 34.

Example 73: JSON photo client

```
import json, socket, camera, appuifw

PHOTO_FILE = u"E:\\Images\\temp.jpg"

def send_photo(name, jpeg):
    msg = {"jpeg": jpeg, "name": name}
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(("192.168.0.2", 9000))
    out = sock.makefile("w")
    out.write(json.write(msg))
    out.close()

while True:
    name = appuifw.query(u"Photo name", "text")
    if not name:
        break
    print "Taking photo.."
    img = camera.take_photo(size = (640, 480))
    img.save(PHOTO_FILE)
    jpeg = file(PHOTO_FILE).read()
    print "Sending photo.."
    send_photo(name, jpeg)
    print "Photo sent ok"
print "Bye!"
```

The function send_photo() is given the JPEG contents of the newly taken photo and the photo's name. These parameters are placed in a dictionary called msg. The dictionary is then encoded to a string using function json.write() and the string is sent over TCP. With JSON, dictionaries provide a handy way to construct protocols of your own. The JSON gateway, which is presented in Section 8.6, provides an example of this.

The server-side code for the photo repository extends Example 69.

Example 74: JSON photo server

```
import SocketServer, json

class Server(SocketServer.TCPServer):
    allow_reuse_address = True

class Handler(SocketServer.StreamRequestHandler):
    def handle(self):
        msg = json.read(self.rfile.read())
        fname = msg["name"] + ".jpg"
        f = file(fname, "w")
        f.write(msg["jpeg"])
        f.close()
        print "Received photo %s (%d bytes)" % (fname, len(msg["jpeg"]))

server = Server(('', 9000), Handler)
print "WAITING FOR NEW CONNECTIONS.."
server.serve_forever()
```

First, the request handler function handle() receives all data from the TCP connection until it is closed by the client. This is done with self.rfile.read(), where self refers to the Handler object and rfile to a file-like object inside it that represents the open TCP connection ('r' in 'rfile' stands for reading).

The client has encoded the data with JSON, so we decode it here with json.read(), which returns the dictionary created by the client. The JPEG data is written to a file using the provided file name. You can check that the image is transferred correctly by opening the JPEG file with your favorite image viewer.

Note that, to be more secure, the server would have to check the file name in msg["name"] to make sure that the client cannot overwrite any previous files or write images to inappropriate locations on your hard disk.

8.4.2 HTTP Server

Some of the following examples require a working web server. If you have already installed a web server and you have a working web back end, such as PHP, Ruby On Rails or TurboGears, to handle requests, you can use it – this applies also to environment D. You should be able to port the server-side examples to your framework of choice without too much effort.

If you do not have such a back end already in place, now is a good time to install one! However, if you want to get into action quickly, a simple web server with a naïve request-handling mechanism is provided. The code relies on two HTTP server modules in Python's standard library, `BaseHTTPServer` and `SimpleHTTPServer`, which implement a rudimentary HTTP server.

The HTTP server example is divided into two parts. Example 75 implements a generic web server that is re-used in later examples. Example 76 shows simple request handlers that demonstrate the server usage. The later examples implement their own functions. You should combine the generic web server (Example 75) and example-specific functions to make the full server.

Example 75: HTTP server

```
import BaseHTTPServer, SimpleHTTPServer, cgi, traceback, json

class Server(BaseHTTPServer.HTTPServer):
    allow_reuse_address = True

class Handler(SimpleHTTPServer. SimpleHTTPRequestHandler):

    def do_POST(self):
        try:
            size = int(self.headers["Content-length"])
            msg = json.read(self.rfile.read(size))
            reply = process_json(msg)
        except:
            self.send_response(500)
            self.end_headers()
            print "Function process_json failed:"
            traceback.print_exc()
            return

        self.send_response(200)
        self.end_headers()
        self.wfile.write(json.write(reply))

    def do_GET(self):
        if '?' in self.path:
            path, query_str = self.path.split("?", 1)
            query = cgi.parse_qs(query_str)
        else:
            path = self.path
            query = {}

        try:
            mime, reply = process_get(path, query)
        except:
            self.send_response(500)
            self.end_headers()
            print >> self.wfile, "Function process_query failed:\n"
            traceback.print_exc(file=self.wfile)
            return
```

```
    self.send_response(200)
    self.send_header("mime-type", mime)
    self.end_headers()
    self.wfile.write(reply)
```

When a web browser requests a URL from this server, the function `do_GET` is called. First, the function checks whether the URL contains any parameters, specified after a question mark. If it does, the parameters are parsed into the dictionary `query` with function `cgi.parse_qs()`. Otherwise, query parameters are initialized with an empty dictionary.

According to the HTTP specification, GET requests must not change the internal state of the server. All information which affects the server is sent in POST requests and encoded in JSON. Replies to the POST requests are also encoded in JSON.

Example 76: Simple request handlers for the HTTP server

```
def process_json(msg):
    return msg

def process_get(path, query):
    return "text/plain", "Echo: path '%s’ and query '%s’" % path, query)

def init_server():
    print "Server starts"

init_server()
httpd = Server((’’, 9000), Handler)
httpd.serve_forever()
```

Each example may perform example-specific initialization in the function `init_server()`.

The requests are handled by the functions `process_get()` and `process_json()`. The function `process_get()` is given two parameters: the requested path from the URL (`path`) and the parsed parameters (`query`). The actual task performed by the function depends on the example. The function returns two values: the MIME type of the response, which tells whether the response contains, for example, HTML, plain text or an image, and the actual response in a string that is sent back to the requester. The requester is typically a web browser.

The function `process_json()` is given the decoded JSON request as a sole parameter. It may return any Python data structure that is then encoded in JSON and returned to the requester. The requester is typically a PyS60 program.

Each of the following examples, which require a web server, provides its own `init_server()`, `process_get()` and `process_json()` functions to handle tasks specific to that particular example. With each example, you should replace these functions in the web server code accordingly.

As with the TCP server, you can pick another port for the server instead of the default, 9000. Once the above code is running on your server, you can try to connect to it with a web browser on your PC and on your phone. The address is of the form ***http://192.168.0.2:9000/test/***, where the IP address corresponds to that of your server. If everything goes well, the browser shows an echo message with the requested path and parameters.

8.4.3 Advanced Topics in Networking

So far, we have focused on using the phone as a network client. Traditionally, this has been the default role for a mobile device, and the operator's data centers have taken care of serving information. Technically, once a mobile device has an IP address there is no reason that it could not act as a server as well. After all, a server is just one of the two equal endpoints in the TCP connection.

Conceptually, reversal of the mobile device's role from an information consumer to a producer is a small revolution. Information produced by your phone is qualitatively different from a typical website. In contrast to a web page, your phone produces information that is bound to time, location and a specific person.

In the next three sections, we see how PyS60 can be used to make the phone a more active information consumer, as well as a producer for the Internet.

8.5 Pushing Data to a Phone

The `socket` module provided by PyS60 closely resembles its sibling on the server side. The module contains functions for listening to a port for incoming TCP connections (functions `bind()`, `listen()` and `accept()`). With these functions you can create a simple TCP server and run it on a phone as you would do on a PC (Figure 8.5).

8.5.1 Drawbacks with using a Phone as a Server

However, three issues tend to make life difficult for phone-based servers:

- Unless the phone stays in one location all the time, its IP address is likely to change when the phone moves from one network to another. Clients must be kept updated on the mobile server's most recent IP address; fortunately, this is possible with several methods.
- Typically, operators do not allow inbound connections over the GSM or 3G network to the phone, because of security and other reasons.

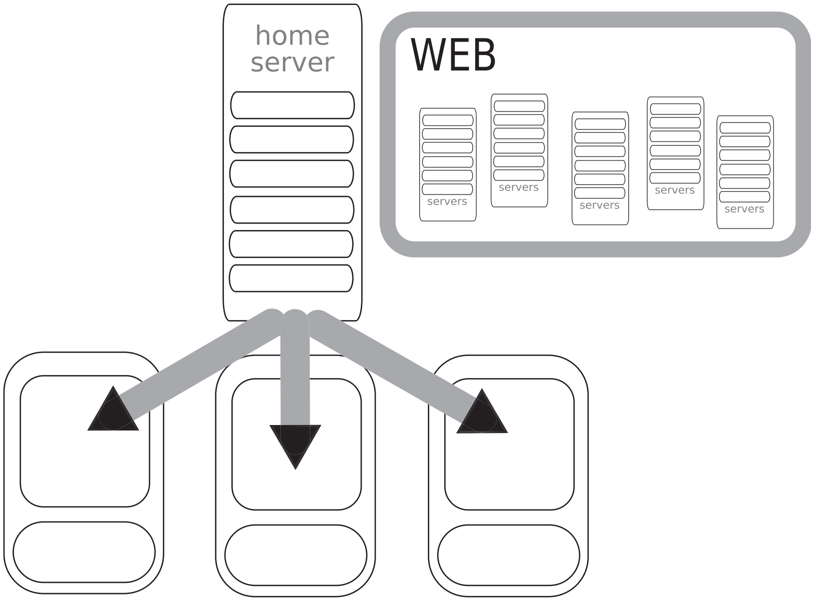


Figure 8.5 Pushing data to a phone

- Even though server-side sockets are supported by PyS60, there is only limited support for handling multiple simultaneous connections on the phone side. Thus, scaling up performance of a server-on-a-phone is not an easy task.

Within a small, local WiFi network these issues might not be issues at all. Thus, it is possible to perform local experimentation on phone-side server software. However, beware that this aspect of PyS60 has received less testing than other, more widely used modules and thus you may find some bugs still lurking in PyS60.

In such a setting, it is important to know the phone's IP address (see Example 77).

Example 77: Phone's IP address

```
import socket
ap_id = socket.select_access_point()
apo = socket.access_point(ap_id)
apo.start()
print "PHONE IP IS", apo.ip()
```

The above three points affect all PyS60 programs that rely on external events that are delivered by the network, including multiplayer games, email clients and real-time chat applications. In all these cases, an external entity, such as a game server, must push some information to the

phone. Since listening for incoming connections on the phone is often not feasible, because of the above issues, it is necessary to consider other approaches.

There are two common workarounds: the phone may poll the server at regular intervals and check whether there is new information to be processed or the phone may open a TCP connection to the server and keep it open so that the server can push information by way of this persistent pipe back to the phone.

The main problem with the former approach is that if the polling interval is too short, say less than a minute, continuous polling is likely to drain the phone's battery. The biggest issue with the latter approach is that it does not work with HTTP. According to the current HTTP specification, HTTP requests always originate from the client, so a server running on the phone cannot initiate the connection.

8.5.2 Voting Service

In this section, the polling approach is demonstrated using HTTP. This example implements a voting service. The server initiates a poll, which consists of several choices. The phone client is allowed to set a vote to one of the choices, within a certain time period that is reserved for voting. Once the voting time has expired, a popup reports the winner on the phone screen. Progress of voting can be followed on a web page in real time.

You should combine the example below with the basic HTTP server in Example 75 to make the full server. These lines should come after the main server. If you are familiar with some web framework, such as Ruby on Rails or Django, you can easily re-implement the server logic in Example 78 using your favorite framework. You can use the phone client in Example 79.

Example 78: Voting server

```
import time, json

def init_server():
    global title, choices, already_voted, started
    started = time.time()
    already_voted = {}
    title = u"What shall we eat?"
    choices = {u"Tacos": 0,\
               u"Pizza": 0,\
               u"Sushi": 0}
    print "Voting starts"

def vote_status():
    voting_closed = time.time() - started > 60
    results = []
```

```
    for choice, count in choices.items():
        results.append((count, choice))
    return voting_closed, max(results)

def process_json(query):
    voting_closed, winner = vote_status()
    if voting_closed:
        return {"closed": True, "winner": winner}

    msg = ""
    if "choice" in query:
        if query["voter"] in already_voted:
            msg = "You have voted already"
        else:
            choices[query["choice"]] += 1
            already_voted[query["voter"]] = True
            msg = "Thank you for your vote!"

    return {"title": title, "winner": winner,\
            "choices": choices, "msg": msg}

def process_get(path, query):
    voting_closed, winner = vote_status()
    msg = "<html><body><h1>Vote: %s</h1><br/>" % title
    for choice, count in choices.items():
        msg += "<b>%s</b> %d<br/>" % (choice, count)

    if voting_closed:
        msg += "<p><h2>Voting closed.</h2></p>"
        msg += "<h1>The winner is: %s</h1>" % winner[1]
    else:
        msg += "<h2>%d seconds until closing</h2>" %\
                       (60 - (time.time() - started))

    return "text/html", "%s</body></html>" % msg

init_server()
httpd = Server(('', 9000), Handler)
httpd.serve_forever()
```

The function `init_server()` initiates the vote. The starting time of the vote is saved in the variable `started`. The dictionary `already_voted` includes the IMEI from the phones which have already cast a vote. The variable `title` gives the name of the vote and the dictionary `choices` holds the candidates with the number of votes they have received thus far.

The function `vote_status()` is used to determine the status of the vote. The variable `voting_closed` is false as long as fewer than 60 seconds have elapsed since the vote started, after that, the variable becomes true. Then, we loop over the `choices` dictionary to determine who is now leading the vote.

We build a list of tuples, so that the number of votes is the first item in the tuple and the candidate name is the second item. From this list, the standard function `max()` returns the tuple in which the first item has the

largest value, hence, the second item in this tuple contains the current leader.

The function `process_json()` handles requests from the phone clients. If the vote has already expired, the function returns immediately and informs the requester about this. Otherwise, if the request `query` includes the keyword 'choice', the client wants to cast a vote. The vote is cast only if this voter has not cast a vote before; that is, the phone IMEI does not exist in the dictionary `already_voted`. The function returns a message that contains all relevant information about the vote and its current status.

The function `process_get()` handles requests from the web browser. Any request produces a web page which shows the current status of the vote. In a more sophisticated web framework, the HTML code would probably be generated using an external template.

The client code is in Example 79. Remember to change the URL to point at your server.

Example 79: Voting client

```
import sysinfo, urllib, json, appuifw, e32

URL = "http://192.168.0.2:9000"
imei = sysinfo.imei()

def json_request(req):
    enc = json.write(req)
    return json.read(urllib.urlopen(URL, enc).read())

def poll_server():
    global voted_already
    res = json_request({"voter":imei})
    votes, winner = res["winner"]

    if "closed" in res:
        appuifw.note(u"Winner is %s with %d votes" % (winner, votes))
        lock.signal()
        return False

    elif not voted_already and "title" in res:
        appuifw.app.title = u"Vote: %s" % res["title"]
        names = []
        for name in res["choices"]:
            names.append(unicode(name))
        idx = appuifw.selection_list(names)
        if idx == None:
            lock.signal()
            return False
        else:
            res = json_request({"voter":imei, "choice":names[idx]})
            appuifw.note(unicode(res["msg"]))
            voted_already = True
            print "Waiting for final results..."
    else:
```

```
        print "%s has most votes (%d) currently" % (winner, votes)

    e32.ao_sleep(5, poll_server)
    return True

voted_already = False
lock = e32.Ao_lock()
print "Contacting server..."
if poll_server():
    lock.wait()
print "Bye!"
```

The client contains two functions: `json_request()`, which encodes a message in JSON, sends it to the server and returns the server reply that is decoded from a JSON message.

The function `poll_server()` contains the program logic. It starts by sending a request to the server that includes the phone's IMEI to identify the phone. The response from the server is assigned to the variable `res`. From the client's point of view, there are three modes:

- The vote is closed, in which case the server response `res` includes the key 'closed'. In this case, the program shows a popup note declaring the winner and exits.
- The vote is ongoing and the user has already cast a vote. In this case, the current status of the vote is printed out on the console.
- The vote is ongoing and the user has not cast a vote yet. In this case, a selection list is shown that presents the choices and lets the user cast a vote. The choice made by the user is then sent to the server.

The current mode is determined by the response message, `res`. If the vote is still ongoing, the `e32.ao_sleep()` function is used to call the function `poll_server()` again after a five-second interval. This way, the client receives the status of the vote from the server almost in real time.

To test this service, start the server software on your server. Once the server is running, you can connect to it with your web browser – the address can be found in the client's `URL` constant. A web page should open that shows the initial status of the vote. Refresh the page every now and then to see the latest status of the vote.

Then, open the client program on your phone and cast a vote for one of the candidates. After this, you can follow progress of voting on the PyS60 interpreter console, until the voting time expires and a popup note is shown that declares the winner. If you have several phones available, any number of phones can cast a vote, but only once per phone.

8.6 Peer-to-Peer Networking

The previous examples have been built on the client–server architecture: the roles of the client and the server have been distinct. The client–server architecture makes sense if the service requires a common 'shared memory' or some other common resource, which is maintained by the server and accessed by the clients. In the voting example, the server maintained the status of voting, which can be seen as an instance of 'shared memory'.

However, think of an instant messaging service or a game of chess between two players. In the former case, the service has no shared state or memory; individual messages are passed back and forth. In the latter case, the state of the shared chess board is easily maintained by both the clients (players) separately. These programs could work without a central server. The communication takes place only between individual phones in a peer-to-peer manner (Figure 8.6).

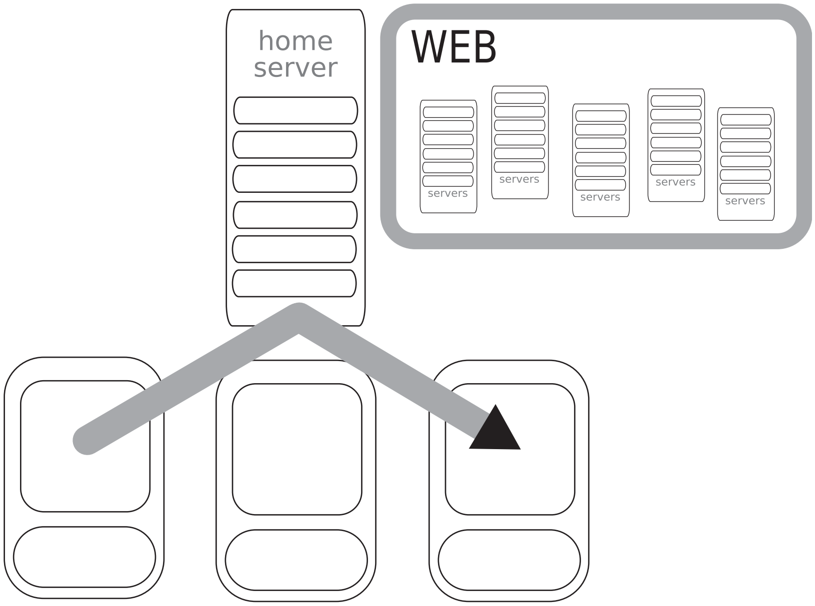


Figure 8.6 Peer-to-peer networking

The issues related to server-side sockets on the phone also apply to peer-to-peer networking, making universal *real peer-to-peer* services difficult to implement. By *real peer-to-peer* networking we mean that the phones are able to communicate directly with each other, without help

from any central server. The Bluetooth chat example in Section 7.3 was a real peer-to-peer application in this sense.

Many WiFi-enabled phones support *ad hoc* networking. In *ad hoc* mode, the phones form a WiFi-network among themselves without help from a base station. In *ad hoc* networks, real peer-to-peer communication is possible but you have to find a way to assign IP addresses to the participating phones. You can use *ad hoc* networks in PyS60 by defining an access point in *ad hoc* mode in the phone's connectivity settings and choosing this access point for networking in PyS60.

If we accept a little help from a server, peer-to-peer networking becomes much more feasible. For instance, this approach is used by Skype and BitTorrent protocols. In the following discussion, we present an approach for peer-to-peer networking with mobile phones that relies on a central server. However, this server is not application-specific. Instead, it is designed to facilitate communication between phones.

Since the server, actually a gateway, is generic in nature, you can use it to build various services in which phones communicate in a peer-to-peer manner. Note that even though Examples 80 and 81 do their jobs properly, probably better alternatives exist for real production environments. For example, have a look at Jabber and the XMPP protocol to see a similar and extensively field-tested, approach.

8.6.1 JSON Gateway

We implement a server (actually, a gateway) that acts as a middle man between phones. Since the gateway runs on your PC (or on some external server), you need to have Python installed there as well. Phones may send any JSON messages to each other, by way of the gateway. With this gateway, it is possible to implement peer-to-peer networking in PyS60.

In technical terms, the gateway works as follows:

- Each client maintains a persistent TCP connection to the central server.
- When a client opens a new connection to the server, it registers itself with a unique name.
- A client sends a message to the server, which includes the recipient's registered name.
- The server pushes the message to the recipient's open TCP connection.

In Section 8.5, we listed three hindrances to building a server on the phone side. This approach neatly solves, or evades, those issues: changing IP addresses are not an issue since addressing works by registered names, not by IP addresses; firewalls and NAT devices are not a problem, since the client always initiates the connection to the server and not other way around; the phone does not have to handle multiple sockets

simultaneously: Messages from all peers are received through a single connection.

We implement a generic JSON message gateway over TCP. You can use this gateway to implement various peer-to-peer applications. If you want to get right into implementing enterprise-level, process-management applications, multiplayer games and collaborative art, you can skip the following explanation of the internals of the gateway. However, the server is not complex and, once you get familiar with it, you can extend it for your own purposes easily.

Conceptually, the code is simple. It works as follows:

```
New Connection:
    Register Client.
    Serve forever:
        Read message, including recipient.
        Client wants to close the connection? Close.
        Otherwise, push the message to the recipient.
```

Unfortunately, the implementation looks a bit more intricate. This is because it has to handle multiple TCP connections simultaneously, in contrast to the previous example servers. Moreover, as this code implements a generic service, instead of a single, simple application, extra attention has been paid to error handling compared to the previous examples.

The gateway uses the following protocol for messaging: each message is a dictionary, encoded in JSON; dictionary keys are strings; keys that have an exclamation mark as the first character are reserved for the gateway protocol. The application is free to use any other strings as keys for its internal communication. The following keys are used by the gateway:

- `!name` – Register the client with this name.
- `!close` – Close the connection.
- `!dst` – Forward message to the named recipient.

The implementation is based on the standard TCP server, but it is spiced up with `SocketServer.ThreadingMixIn`, which modifies the server behavior so that each client is handled in a separate thread of execution. The result is that multiple connections can be handled simultaneously, but we must be extra careful that the threads do not mess up each other.

The gateway code is divided into two parts. You should combine them to make the final application. The first part, Example 80, introduces some core data structures and the client registration mechanism.

Example 80: Generic JSON gateway (1/2)

```
import SocketServer, threading, json

conn = {}
```

```
conn_lock = threading.Lock()

class ThreadingServer(SocketServer.ThreadingMixIn,
                      SocketServer.TCPServer):
    allow_reuse_address = True

class Handler(SocketServer.StreamRequestHandler):
    def handle(self):
        print "A new client connected", self.client_address
        msg = json.read_stream(self.rfile)
        if "!name" in msg:
            name = msg["!name"]
            wlock = threading.Lock()
            conn_lock.acquire()
            conn[name] = (wlock, self.wfile)
            conn_lock.release()
            print "Client registered (%s)" % name
            reply = {"ok": u"registered"}
            self.wfile.write(json.write(reply))
            self.wfile.flush()
        else:
            reply = {"err": u"invalid name"}
            self.wfile.write(json.write(reply))
            return
        handle_connection(self, name)
```

The core of the server is the dictionary `conn`, which is shared by all handler threads. It maps registered names to the corresponding connections. Any phone that wants to participate in peer-to-peer networking must open a connection and register a name with the server.

Since multiple threads may add and remove keys from the dictionary in parallel, its integrity must be protected with a lock (`conn_lock`). A thread may access the dictionary only when it has acquired the lock. To make the access mutually exclusive, the lock can be held by only one thread at a time. Do not worry if locking issues confuse you. It is a notoriously difficult subject.

When a new client connects to the gateway, it first sends a message containing the key '!name' to it. This key maps to a name that identifies this phone. Note that the gateway does not check whether another phone with the same name has registered already. The gateway just blindly replaces the old connection with a new one. The gateway sends a reply containing the key 'ok' to the client if registration was successful.

This means that anyone can hijack an existing connection to the gateway and pretend to be the original phone. If the gateway was to be used in an uncontrolled setting, some other approach for handling already registered names would need to be implemented. This protocol is not meant for secure communication.

Example 81: Generic JSON gateway (2/2)

```
def handle_connection(self, name):
```

```
    while True:
        try:
            msg = json.read_stream(self.rfile)
        except:
            msg = {"!close": True}

        if "!close" in msg:
            print "Client exits (%s): %s" % name, self.client_address)
            conn_lock.acquire()
            if name in conn:
                del conn[name]
            conn_lock.release()
            break
        elif "!dst" in msg:
            wfile = None
            conn_lock.acquire()
            if msg["!dst"] in conn:
                 wlock, wfile = conn[msg["!dst"]]
            conn_lock.release()
            if wfile:
                wlock.acquire()
                try:
                    wfile.write(json.write(msg))
                    wfile.flush()
                finally:
                    wlock.release()

server = ThreadingServer(('', 9000), Handler)
print "JSON gateway is running!"
print "Waiting for new clients..."
server.serve_forever()
```

The second part of the gateway contains the function `handle_connection()`, which is used to handle a connection between a single phone and the gateway. Note that, because of threading, many `handle_connection()` functions are active simultaneously in the gateway, each handling an individual phone.

Multiple clients may want to push a message to the same client at the same time. Since separate messages must not be mixed up in the connection, only one thread may write to a connection at once. This is ensured by a connection-specific lock, `wlock`.

Note that messages are read with the function `json.read_stream` instead of `json.read`, which has been used in the previous examples. This function parses a message byte by byte from the connection and blocks until it has completely read one message. With this function, the client can use a single permanent connection to send and receive multiple messages.

The server is oblivious to errors in message sending. If the recipient moves away, say, from a WiFi hotspot while a message is being sent, the message is quietly lost. This also happens if the recipient for a message is unknown. Thus, any application using this gateway must understand that any message may be silently lost.

If an error occurs while a message is being read, the gateway deregisters the client, assuming that it has lost its network connection and will reconnect when a network becomes available again. However, note that if another client registers to the gateway using the same name, subsequent messages will be routed to this new, possibly malicious, client.

This server is run in a similar way to the previous example servers, such as the simple TCP server (Example 69). Naturally, you need to start the server before any phones can start to communicate with each other.

The gateway server can easily be extended with new features. For example, it might be useful to be able to distribute a message to multiple recipients. Broadcasting like this should not be difficult to implement: Just add a new protocol key, say `"!broad"` and loop through the `conn` dictionary, pushing the message to each active connection.

8.6.2 Instant Messenger

A natural application for peer-to-peer communication is instant messaging. With this application, any number of phones can take and send photos and short text messages to each other over the Internet or a local wireless network. If all participants have WiFi-capable phones and they use a free wireless LAN, you can use this application to share an unlimited number of photos and text messages with your friends at no cost!

Technically, this example shows how to build applications on top of the JSON gateway. The example presents a generic client-side counterpart for the JSON gateway, which can easily be used by peer-to-peer applications of your own.

This code also exemplifies how to handle network events in an asynchronous manner in PyS60 using threads. We do not explain threading in detail in this book, but you can use the following functions in applications of your own. As a non-trivial exercise, you could make the Bluetooth chat application in Section 7.3 asynchronous by modifying the instant messenger code to use Bluetooth for communication.

The example is divided into three parts. Example 82 contains the reusable communication infrastructure. Examples 83 and 84 are related to the user interface of the instant messaging application. As usual, you should combine these three parts to make the final application.

Example 82: Instant messenger (1/3)

```
import appuifw, e32, camera, thread, socket, json, graphics
SERVER = ("192.168.0.2", 9000)

def send_message(msg):
    global to_server
    try:
```

```
        to_server.write(json.write(msg))
        to_server.flush()
        thread_handle_message({"note": u"Message sent!"})
    except Exception, ex:
        print "Connection error", ex
        to_server = None

def read_message():
    global to_server
    try:
        msg = json.read_stream(to_server)
        thread_handle_message(msg)
    except Exception:
        print "Broken connection"
        to_server = None

def connect():
    global to_server, keep_talking, conn
    conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    conn.connect(SERVER)
    to_server = conn.makefile("rw", 8192)
    send_message({"!name": me})
    ret = json.read_stream(to_server)
    if "err" in ret:
        thread_handle_message({"note": u"Login failed"})
        thread_show_note(u"Login failed: %s" % ret["err"], "error")
        keep_talking = False
        raise "Login failed"
    else:
        thread_handle_message({"note": u"Login ok"})

def communicate():
    global to_server, keep_talking, thread_send_message, app_lock
    thread_send_message = e32.ao_callgate(send_message)
    to_server = None
    while keep_talking:
        if to_server:
            read_message()
        else:
            try:
                connect()
                thread_handle_message({"note": u"Waiting for\
                                              messages..."})
            except:
                print "Could not connect to server"
                to_server = None
                if keep_talking:
                    e32.ao_sleep(10)
    if conn:
        conn.close()
    if to_server:
        to_server.close()
```

The combination of an interactive user interface and asynchronous, two-way communication with a server is a tricky one. The application must listen to both the user and the server simultaneously, as either of them may need attention at any time.

To fulfill this need, the communication infrastructure is run in a separate thread from the UI. Thus, this example uses two threads: the main thread that handles the UI and the communication thread that communicates with the JSON gateway. As with the gateway code, above, extra attention is needed to keep the threads from messing each other up.

The core of the communication thread is the function `communicate()`. It keeps waiting for incoming messages in an infinite loop until the application quits. Each iteration begins with a check that the connection is still alive. If it is not, the function `connect()` is called which tries to register the client to the gateway. If the registration fails, it will be tried again after a ten-second delay. Remember to change the `SERVER` constant in Example 82 to point at the actual server that is running the JSON gateway.

Once the connection has been set up, the function `read_message()` is used to receive a new message. When a new message arrives, it is processed by an application-specific function, `thread_handle_message()`. This function is actually a wrapper, produced by the `e32.ao_callgate()` function, for the function `handle_message()`.

A wrapper like this is needed whenever a thread must call a function that accesses low-level resources, such as network connections (sockets) or the user interface, which have been initialized by another thread.

In this case, `handle_message()` is used to access some UI resources, for instance, `appuifw.note()`, which have been initialized by the main thread. Similarly, the function `send_message()` is wrapped to the function `thread_send_message()` which lets the main thread access the connection to the server, which is owned by the communication thread.

Sharing resources between threads is a complicated matter. Often, it is a good idea to avoid using threads in the first place because of these difficulties. If your application really needs them, the best idea is to re-use designs that have been proven to work before.

To re-use this communication infrastructure, namely functions `read_message()`, `send_message()`, `connect()` and `communicate()`, in another application, you have to replace the function `handle_message()` with an application-specific handler.

Example 83 shows the message handler function `handle_message()` for the instant messenger.

Example 83: Instant messenger (2/3)

```
def show_photo(jpeg_data):
    global img
    f = file("E:\\Images\\msg.jpg", "w")
    f.write(jpeg_data)
    f.close()
    img = graphics.Image.open("E:\\Images\\msg.jpg")

def handle_message(msg):
    global text, note
```

```
    if "photo" in msg:
        show_photo(msg["photo"])
        text = {"from": msg["from"], "txt": ""}
    elif "txt" in msg:
        text = msg
    elif "note" in msg:
        note = msg["note"]
    redraw(None)

def send_photo():
    handle_message({"note": u"Taking photo..."})
    dst = appuifw.query(u"To", "text")
    img = camera.take_photo(size = (640, 480))
    img = img.resize((320, 240))
    img.save("E:\\Images\\temp.jpg")
    jpeg = file("E:\\Images\\temp.jpg").read()
    handle_message({"note": u"Sending photo..."})
    thread_send_message({"!dst": dst, "photo": jpeg, "from": me})

def send_text():
    resp = appuifw.multi_query(u"To", u"Message")
    if resp:
        dst, txt = resp
        thread_send_message({"!dst": dst, "txt": txt, "from": me})
```

This part of the application contains many functions related to graphics and the user interface that are already familiar to you from previous examples. The application has to handle four tasks:

- The user may take and send a photo through the 'Send photo' menu item. This task is handled by the function `send_photo()`.
- The user may send a text message through the 'Send text' menu item. This task is handled by the function `send_text()`.
- The user may receive a photo from another user. The incoming message is first handled in `handle_message()` which calls the function `show_photo()` to prepare the received photo for showing.
- The user may receive a text from another user. The incoming message is first handled in `handle_message()`, as above, and the function `redraw()` shows the received message on the canvas.

Note that both the send functions use the function `thread_send_message()` to pass a new message to the networking thread. This function is a wrapper for the function `send_message()` which actually sends the message to the gateway. As required by our JSON gateway, the `"!dst"` key is used to define the recipient's name in a message.

Example 84: Instant messenger (3/3)

```
def quit():
    global keep_talking, to_server, conn
```

```
    keep_talking = False
    thread_send_message({"!close": True})
    app_lock.signal()

def redraw(rect):
    RED = (255, 0, 0)
    GREEN = (0, 255, 0)
    BLUE = (0, 0, 255)

    canvas.clear((255, 255, 255))
    if img:
        canvas.blit(img, scale = 1)
    if note:
        canvas.text((10, canvas.size[1] - 30), note,\
                          fill = RED, font = "title")
    if text:
        canvas.text((10, 80), u"From: %s" % text["from"],\
                             fill = GREEN, font = "title")
        canvas.text((10, 110), unicode(text["txt"]),\
                         fill = BLUE, font = "title")

img = text = note = None
keep_talking = True
thread_handle_message = e32.ao_callgate(handle_message)
appuifw.app.exit_key_handler = quit
appuifw.app.title = u"Instant Messenger"
appuifw.app.menu = [(u"Send Photo", send_photo),\
                       (u"Send Text", send_text)]

canvas = appuifw.Canvas(redraw_callback = redraw)
appuifw.app.body = canvas

me = appuifw.query(u"Login name", "text")
handle_message({"note": u"Contacting server..."})
if me:
    app_lock = e32.Ao_lock()
    thread.start_new_thread(communicate, ())
    if keep_talking:
        app_lock.wait()
```

In this last part of the application, there are two new functions, which have not been used earlier in this book. The function `e32.ao_callgate()` was discussed above. The function `thread.start_new_thread()` is used to start the function `communicate()` in another thread of execution.

This means that the `while` loop in the function `communicate()`, which is responsible for receiving incoming messages, keeps on going in the background although events related to the user interface are handled in the main thread.

You can test this application by sending messages to yourself. Just log in with an arbitrary name, for instance 'Matt', choose 'Send Photo' in the application menu and as the recipient, write 'Matt'. Almost immediately, after the message has made a round-trip to the JSON gateway, the photo should appear on your phone's screen.

8.7 Using a Phone as a Web Service

Accessing the phone's resources by way of the web is a 'bleeding-edge' approach that might be used as a springboard for many kinds of unprecedented applications.

We have already seen some examples where a PyS60 program fetches some information from the web. Many more examples along these lines follow in Chapter 9. In this section, however, we consider the opposite case: the phone could serve information to the web (see Figure 8.7).

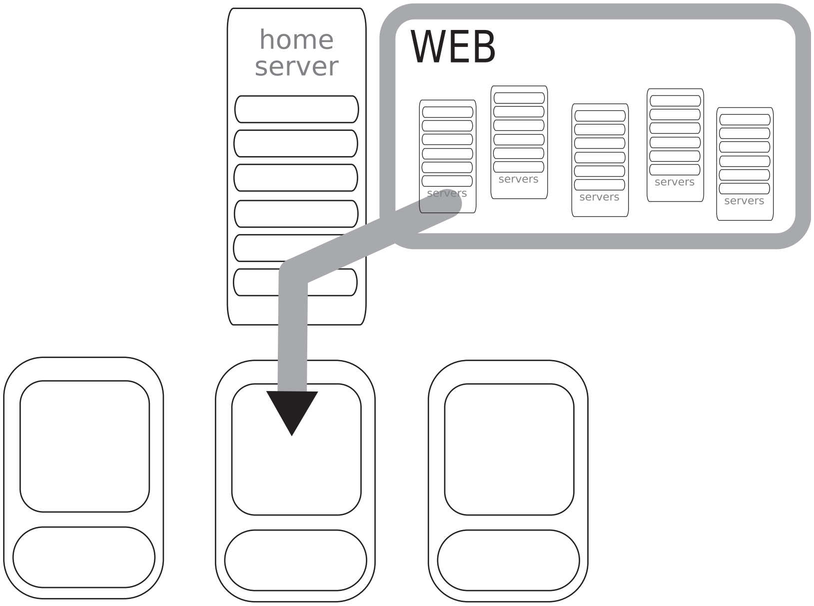


Figure 8.7 Phone providing a web service

Think about various possibilities: for example, you could take a photo with someone else's camera phone from a web page, you could request the phone's holder in real time to take a photo for you or your mobile website could list people around you at that moment, based on Bluetooth scanning.

Apache, the world's most popular web server, has been ported to the S60 platform so one can experiment with scenarios like this. You could use Mobile Web Server (MWS) to implement the above scenarios. You can even use PyS60 to write extensions and request handlers for the server. You can find the MWS home page at ***http://opensource.nokia.com/projects/mobile-web-server/***.

Here, however, we show you how to serve resources of your mobile phone to the web in pure PyS60. Compared to MWS, this approach is more flexible since you can control every part of the server. On the other

hand, in some settings this approach may require much more work, since you have to control every part of the server. Anyway, it is good to know that the basic task is not really complex, as Example 86 shows.

The 'Phone as a web server' scenario is not different from the other phone-server scenarios which have been discussed in Sections 8.5 and 8.6. Generally speaking, one cannot connect to the phone directly from the Internet. There are two options: either use a web proxy, to which the phone pushes information or have a gateway which routes HTTP requests to the phone, in a similar way to the JSON gateway.

We give an example of the proxy approach. A proxy server receives HTTP requests from the web and saves them until the phone polls the server for new requests. The proxy caches responses sent by the phone, so the web clients get a response without any delay.

Since web clients are not directly in contact with the phone, this approach scales well. Also, the proxy can handle a cluster of phones in the background. Requests can be directed to a specific phone based on the URL or the proxy can balance requests between the phones.

It is even possible to construct an on-demand web service, where only those phones which have access to a requested resource, for instance, those in a specific location, respond to the requests. Although this might sound extremely sophisticated, you have already learnt enough to do it by yourself, based on Example 85 and positioning techniques presented in Section 6.4.

The proxy is based on the simple HTTP server that is described in Section 8.4.2. As before, you should be able to port this example to any web framework of your choice with relative ease, instead of using our toy HTTP server. The web server functions are elegantly simple as all the processing is done on the phone side.

Example 85: Phone–web proxy

```
def init_server():
    global phone_req, cache
    phone_req = {}
    cache = {}

def process_json(msg):
    global cache
    cache = msg
    return phone_req

def process_get(path, query):
    phone_req[path] = True
    if path in cache:
        return cache[path]
    return "text/plain",\
        "Your request is being processed."\
        "Reload the page after a while."
```

```
init_server()
httpd = Server(('', 9000), Handler)
httpd.serve_forever()
```

The phone both delivers responses to previous requests and fetches new ones using JSON messages that are handled by the `process_json()` function. Requests by the web browser are handled in the `process_get()` function. If the requested resource is not available in the cache, a brief message is returned that informs the user about this.

You should combine this example with the basic HTTP server in Example 75 to make the full proxy server. These lines should come after the main server.

On the phone side, we make some resources of the phone available to the proxy, namely the camera, screenshots and the battery indicator. In other words, you can use the phone's camera and see its current screen and the battery level on the web browser! It is really straightforward to add more resources to be shared by the phone.

If this sounds unbelievable to you, just see Figure 8.8. The screenshot showing on the browser is not a static image, but is generated by this example on the fly as requested by the web browser. Amusingly, the screenshot captured some log lines printed out on the console which were caused by the screenshot request itself.



Figure 8.8 The phone's screen as viewed in a web browser

Example 86: Phone–web server

```
import e32, json, camera, graphics, sysinfo, urllib

URL = "http://192.168.0.2:9000"

def json_request(req):
    enc = json.write(req)
    return json.read(urllib.urlopen(URL, enc).read())

def take_photo():
    img = camera.take_photo(size = (640, 480))
    img.save("E:\\Images\\temp.jpg")
    return file("E:\\Images\\temp.jpg").read()

def screenshot():
    img = graphics.screenshot()
    img.save("E:\\Images\\temp.jpg")
    return file("E:\\Images\\temp.jpg").read()

go_on = True
msg = {}
print "Web service starts..."
while go_on:
    ret = {}
    for path in json_request(msg):
        print "Requesting", path
        if path == "/camera.jpg":
            ret[path] = ("image/jpeg", take_photo())
        elif path == "/screenshot.jpg":
            ret[path] = ("image/jpeg", screenshot())
        elif path == "/battery":
            ret[path] = ("text/plain",\
                         "Current battery level is %d" %
                                 sysinfo.battery())
        elif path == "/exit":
            go_on = False
        else:
            ret[path] = ("text/plain", "known resource")
    msg = ret
    e32.ao_sleep(5)
print "Bye!"
```

Again, make sure that the constant `URL` refers to your actual server. These resources are mapped to logical URLs, '/camera.jpg', '/screenshot.jpg' and '/battery' – see the navigation bar in Figure 8.8 to see one of the URLs in practice. The resulting code is surprisingly simple. It can be made even simpler using Python's introspection features, as we will show in Section 10.2.

Note that when testing this example, you may need to request a URL on the phone, for example, ***http://192.168.0.2:9000/camera.jpg***, several times before you see the actual result on the browser. The server exits when the 'exit' resource is requested.

This example makes a great showcase of Python's power and expressiveness: in fewer than 120 lines of code, we have implemented a working

web server and a proxy, and turned a mobile phone into a web service which features a webcam, a screen capturer and a battery-level indicator!

8.8 Summary

This chapter covered the basics of modern networking in a nutshell. After reading this chapter, you should be prepared to build innovative mobile-device-centric, networked programs and services.

This chapter contains a lot of information and you may feel it hard to digest everything at once. The best approach is to start experimenting, exploring and tinkering around with your own ideas. If you need more information about any subject in this chapter, the web is your friend.

The chapter started with examples of TCP, HTTP and JSON clients which requested information from the outside world for the mobile phone. The latter part of the chapter focused on using the phone as a server. Techniques of polling, persistent TCP connections, gateways and web proxies were introduced to aid server development on the phone side.

This chapter presents several working example applications: a camera which saves photos to a server, a voting server with mobile and web interfaces, a JSON gateway for peer-to-peer communication, an instant messenger and a web service which lets you control your phone from a web browser. These examples may be useful after some polishing or you may use them as the basis for your own applications.

9

Web Services

In Chapter 8, we showed how you can use PyS60 to build network clients and servers of your own. In this chapter, we show how to tap into services provided by others. These two approaches are becoming more and more complementary. Often the most interesting results emerge when you combine something of your own, such as your current location, with some external information, such as Google Maps.

Information available in ordinary websites is primarily prepared for human consumption. Data is embedded in elaborate visual layouts that look pleasing to the eye but from which it is difficult to extract the data programmatically. Technically, it would be possible to retrieve any web page using, for example, the `urllib.urlopen()` function but parsing the desired data from the midst of complex HTML code is a tedious and error-prone task. Luckily, some Python libraries, such as Beautiful Soup, exist to make this task easier.

However, many websites and web application providers have understood that they can increase the value of their product by making it easily accessible to other programs, as well as human users. Increasingly often, they are starting to provide another view to their service that can be used to retrieve the pure data easily without any decoration. This interface is often called a *web service* or a *web Application Programming Interface* (*web API*).

For example, Amazon, Google, Yahoo!, Flickr, Facebook and most of the blog engines provide a web API through which their service, or some parts of it, can be used programmatically. Typically, the web API is made freely available for non-commercial use only. You should always read the terms of use carefully before using any web API in your program.

Note that practically all services require that you apply for an *application key* to use the service. The key is used by the service to identify the origin of third-party requests and to control the number of requests made

by an external application. Typically, you receive the key immediately when you register a web API account.

9.1 Basic Principles

You can think of a web service as an ordinary Python module whose functions are just called in a particular way using `urllib`. As with ordinary modules, you have to see the API documentation to understand how to use the related functions. However, whereas the API of an ordinary Python module stays constant, even in different versions of Python, a web service provider may change the web API any time without prior notice. Because of this, we focus mostly on the basic principles of the Web service usage here, instead of explaining the current web APIs in detail.

To use a web service, you typically have to go through the following steps:

1. Go to the service provider's website and find the web API or developer documentation. Some well-known web APIs can be found at following addresses:
   - Amazon: ***http://aws.amazon.com***
   - Flickr: ***www.flickr.com/services/api***
   - Google: ***http://code.google.com/apis***
   - Yahoo!: ***http://developer.yahoo.com/python***
2. Find documentation for the functionality you need.
3. Find out which protocol the service uses on top of HTTP. Many services support several different protocols. The most typical alternatives are JSON, XML or plain HTTP. Some services even provide a custom module for Python that hides requests to the web service behind normal Python function calls. However, some of these modules use new features of Python which are not yet supported by Python for S60.
4. Find out what kind of input data and parameters the service expects and how the output looks.
5. Add a function to your program that prepares data for the service and handles the output.
6. Add calls to the web service with the `urllib.urlopen()` function.
7. See how it works!

Web services are built on top of the techniques that were presented in Chapter 8. They are accessed over HTTP and responses are typically encoded either in JSON or XML. Example 71 in Chapter 8 demonstrated this approach.

Some services follow REST principles. These services let you access the data through simple URL addresses, not unlike ordinary web pages. In Chapter 8, we built a RESTful web service of our own in Example 86. It made some resources of your mobile phone accessible to the network through simple URLs, such as ***http://192.168.0.2/battery/***. REST is probably the simplest request format to use since it requires only familiar `urllib.urlopen()` calls.

This chapter presents three fully working and useful applications that are based on three different web services. Each of the examples makes simple requests to the service and gets the response either in XML or JSON. Since many modern web services support an API like this, you can easily apply the following ideas to your own projects.

Note that the web APIs are subject to frequent changes. It may happen that the APIs used by the following examples have been changed since the publication of this book. In this case, you should be able to find out the new syntax in the service's API documentation and modify the example accordingly.

9.2 MopyMaps! Mobile Yahoo! Maps

MopyMaps! is a mobile map explorer. It uses the Yahoo! Maps Web API to receive map images given a desired location. MopyMaps! supports vertical and horizontal panning of received map images, so you can explore your surroundings smoothly on the small screen of a mobile phone. MopyMaps! squeezes a full world atlas into a pocket – all in fewer than 100 lines of Python!

The Map Image API provided by Yahoo! Maps is extremely simple to use. You can give a partial address of a desired location in free-form text and the system infers the most suitable map image for you. You can specify the desired map size, zoom level and radius for the image. Full documentation for the Map Image API is available at ***http://developer.yahoo.com/maps***.

Instead of the address, you can specify the longitude and latitude of the desired location. If you have a GPS receiver, you can combine MopyMaps! with the GPS reader program that was presented earlier to receive a map of your current surroundings automatically.

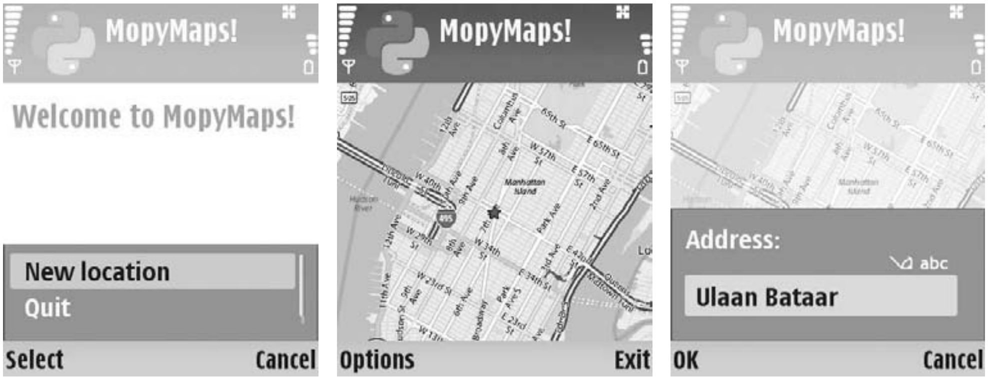
You need to get an Application ID from Yahoo! to use the service. Go to ***http://developer.yahoo.com*** and choose the link 'Get an Application ID'. Once you have filled in the developer registration form, you are given a long data string that is your Application ID. Save this string for future use.

The MopyMaps! source code is divided into three parts that should be combined into one file to form the full application. The parts cover the following functionalities:

- constants and result parsing

- fetching map images from Yahoo! Maps
- user interface functions.

MopyMaps! uses the standard `appuifw` UI framework (Figure 9.1). The map location can be changed by way of a menu item (`new_map`) that triggers a request to the map service. The application body contains a `Canvas` object to which the received map image is plotted (`handle_redraw`). The image can be moved around using the arrow keys (`handle_keys`). Messages related to map loading and possible error conditions are drawn on the canvas as well (`show_text`).



(a) (b) (c)

Figure 9.1 MopyMaps! (a) Welcome screen, (b) map display and (c) address dialog

Let's dive into the code!

9.2.1 Constants and Result Parsing

Example 87: MopyMaps! (1/3)

```
import urllib, appuifw, e32, graphics, key_codes, os, os.path

APP_ID = "reBqRjOdK4E3aKfuRioOj3459Kmas_ENg7!!"
MAP_URL = "http://local.yahooapis.com/MapsService/V1/mapImage?"
MAP_FILE = u"E:\\Images\\mopymap.png"

if not os.path.exists("E:\\Images"):
    os.makedirs("E:\\Images")

def naive_xml_parser(key, xml):
    key = key.lower()
    for tag in xml.split("<"):
        tokens = tag.split()
        if tokens and tokens[0].lower().startswith(key):
            return tag.split(">")[1].strip()
    return None
```

Three constants are used by the application: APP_ID is the application ID that you got from the Yahoo! developer site. Replace the string in the example with your personal ID. MAP_URL is the URL of the service. You can check the current address at the API documentation page, in case the address has changed since the book's publication. MAP_FILE is a temporary file name for the map images.

The Yahoo! service does not return a map image directly after a request. Instead, it returns a small XML message that contains a temporary URL where the image can be retrieved (or an error message, if a suitable map could not be found). The responses look like this:

```
<?xml version="1.0"?>
<Result xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
http://gws.maps.yahoo.com/mapimage?MAPDATA=LkE5ukmEyGeRhYl...
</Result>
<!-- ws01.search.scd.yahoo.com uncompressed -->
```

Officially, it should be the job of an XML parser to parse and convert a string like this to a convenient data structure. However, if we know that the messages are going to be really simple, as in this case, we may use a lighter solution instead of a fully fledged XML parser.

Function naive_xml_parser() is such a solution. Given a simple string of XML, xml, it returns the contents of the first tag whose name begins with key. In MopyMaps! this function is used to extract from the Result tag either the map URL or an error message, if the URL could not be found.

The function works by first splitting the string using < characters as delimiters, which correspond to the beginnings of XML tags. Then, the beginning of each token is compared with the given key to find the matching tag. Matching is case insensitive. The function returns None if no matching tag was found.

You should be extremely careful when using an *ad hoc* solution like this. Even a minor change in the message format may cause the function to return incorrect results. A real XML parser is probably a more suitable solution for any widely distributed application. The function naive_xml_parser() is used here for its brevity and simplicity, not because of its robustness.

9.2.2 Fetching Map Images from Yahoo! Maps

Example 88 contains the core of the MopyMaps! application. It fetches map images from Yahoo! Maps. The functionality for retrieving the images is divided between two functions: new_map() asks the user for a new address and requests the corresponding map image URL from the service; load_image() is then responsible for retrieving the actual image data, given the new URL. It also resets the map position (variables map_x and

map_y), clears any status messages (status) and loads the new map image into a global Image object, mapimg, that is used to hold the displayed map.

As mentioned above, Yahoo! Maps provides a simple interface based on plain HTTP. This means that request parameters are provided to the service in the URL. There is no need to construct special request messages. This is good news for us.

Example 88: MopyMaps! (2/3)

```
def new_map():
    addr = appuifw.query(u"Address:", "text")
    if not addr:
        return
    params = {"location": addr,
              "appid": APP_ID,
              "image_type": "png",
              "image_height": "600",
              "image_width": "600",
              "zoom": "6"}
    show_text(u"Loading map...")
    try:
        url = MAP_URL + urllib.urlencode(params)
        res = urllib.urlopen(url).read()
    except:
        show_text(u"Network error")
        return

    img_url = naive_xml_parser("result", res)
    if img_url:
        show_text(u"Loading map......")
        load_image(img_url)
        handle_redraw(canvas.size)
    else:
        msg = naive_xml_parser("message", res)
        show_text(u"%s" % msg)

def load_image(url):
    global mapimg, map_x, map_y, status
    res = urllib.urlopen(url).read()
    f = file(MAP_FILE, "w")
    f.write(res)
    f.close()
    mapimg = graphics.Image.open(MAP_FILE)
    map_x = mapimg.size[0] / 2 - canvas.size[0] / 2
    map_y = mapimg.size[1] / 2 - canvas.size[1] / 2
    status = None
```

Since we use the standard urllib.urlencode() function to construct the request URL, parameters must be given in a dictionary. The dictionary params specifies the new map:

- location is the address string given by the user.
- appid is the Yahoo! Application ID.

- image_type specifies that we want the image in PNG format.
- image_height and image_width give the requested map size in pixels – you may increase these values if you want a larger area for scrolling.
- zoom specifies the map scale in a range from 1 (street level) to 12 (country level).

You could enhance the application by letting the user modify the zoom level. You can find the full list of supported parameters in the Yahoo! Map Image API documentation.

After the request parameters have been encoded to the URL string url, we send the request to the service with a familiar urllib.urlopen() call. If any exceptions occur during the call, an error message is shown and the function returns. Otherwise we receive a response XML message, which contains the actual image URL inside a result tag. If our simple XML parser does not find the result tag, we try to find an error message inside a message tag instead, which is then shown on the screen.

If the image URL is found, that is, img_url is not None, the function load_image is called to load the actual image. This function retrieves the image data from the result URL, writes the image to a local file and loads it to an image object mapimg. Since the service returns the maps centered at the requested address, we initialize the map position (map_x and map_y) to the middle of the canvas.

The map loading progress is shown by an increasing number of dots after the 'Loading map' text. Three dots are shown when the map image URL is first requested. Six dots are shown when we start to load the actual image.

9.2.3 User Interface Functions

The last part of MopyMaps! is presented in Example 89. This part puts the pieces together by constructing the application UI. The core functionality here is formed by the two canvas callbacks, handle_redraw() and handle_keys() whose roles are already familiar from Chapter 5.

Example 89: MopyMaps! (3/3)

```
def show_text(txt):
    global status
    status = txt
    handle_redraw(canvas.size)

def handle_redraw(rect):
    if mapimg:
        canvas.blit(mapimg, target=(0, 0), source=(map_x, map_y))
    else:
```

```
        canvas.clear((255, 255, 255))
    if status:
        canvas.text((10, 50), status, fill=(0, 0, 255), font="title")

def handle_keys(event):
    global map_x, map_y
    if event['keycode'] == key_codes.EKeyLeftArrow:
        map_x -= 10
    elif event['keycode'] == key_codes.EKeyRightArrow:
        map_x += 10
    elif event['keycode'] == key_codes.EKeyUpArrow:
        map_y -= 10
    elif event['keycode'] == key_codes.EKeyDownArrow:
        map_y += 10
    handle_redraw(canvas.size)

def quit():
  app_lock.signal()

map_x = map_y = 0
mapimg = status = None

appuifw.app.exit_key_handler = quit
appuifw.app.title = u"MopyMaps!"
appuifw.app.menu = [(u"New location", new_map), (u"Quit", quit)]

canvas = appuifw.Canvas(redraw_callback = handle_redraw,
                        event_callback = handle_keys)
appuifw.app.body = canvas

show_text(u"Welcome to MopyMaps!")
app_lock = e32.Ao_lock()
app_lock.wait()
```

Arrow keys are used to move the map image on the screen. This is not too difficult, since we can specify the source coordinates for the `canvas.blit()` function, which correspond to the upper left corner of the map image. By modifying the corner coordinates `map_x` and `map_y`, we can alter the part of the map shown on the canvas. Note that no checks are made that the map image fully fills the canvas. You can see the results of this easily by scrolling far enough on the map. If the result is not pleasing to your eye, you can ensure that the boundaries are not exceeded by adding a few additional `if` statements in the `handle_keys()` function.

The function `handle_redraw()` takes care of drawing the map and the status message – if the corresponding variables are not empty. It's a good habit to handle all drawing in a centralized manner in the redraw callback. This makes sure that the screen is redrawn correctly after, say, a screensaver or an external dialog has been drawn on top of our application. The status message is updated by way of the `show_text()` function, which is just a shorthand for the required three lines of code.

MopyMaps! could be extended easily. As mentioned above, the map could be retrieved based on the current GPS coordinates instead of

an address. The map could be zoomable or alternatively the user could specify how many miles the map should cover (see the `radius` parameter in the API).

You can combine maps with some other information source. For example, Yahoo! provides a traffic API for some parts of the world which shows real-time traffic information that can be plotted on the map. It also provides a search API that lists establishments in proximity to a given location – maybe you can come up with a trendy mashup that combines the maps with the event information that is used in Section 9.3.

9.3 EventFu: Finding Eventful Events

Have you ever been in a foreign city without anything to do? Have you ever seen an advertisement about a concert but forgotten where it will happen and when? EventFu is here to help: it connects to the Eventful service at ***http://eventful.com/*** to retrieve information and keeps you updated about events that are interesting to you.

Eventful is an event database with an extensive web API which can be found at ***http://api.eventful.com***. EventFu uses only one of the provided functions, event search, and shows a list of events that match the user's preferences. The list is updated at regular intervals so new events appear on the list after a short delay once they have been listed in the service.

Requests are made to the Eventful API using a simple URL-encoded interface similarly to the previous example. However, Eventful supports responses in JSON, so the results can be converted directly to a convenient data structure. To use the service, you have to request an application key from Eventful at ***http://api.eventful.com/keys/new***. The key is a long random string not unlike the Yahoo! Application ID.

Note that the `json` module is needed in the example. It is not included in the standard PyS60 distribution; see Section 8.2.2 for how to install this module.

EventFu contains several features that are worth noticing:

- It presents a straightforward mapping between the UI and the web API – for example, the event description form is constructed on the fly based on the Eventful response fields.
- It shows how to update data from an external source continuously at regular intervals.
- It gives a practical example of how to save user preferences to a local database and how to use the phone's default browser to show HTML pages.
- It lets the user select the default access point for networking – a crucial feature when the application must use the network without user intervention.

The EventFu source code is divided into five parts. Again, the parts should be combined into one file to form the full application. The parts cover the following functionalities:

- constants and preferences form
- storing preferences
- event form and event description
- updating events using the Eventful API
- access point dialog and user interface functions.

EventFu uses the standard UI extensively. The main view is based on an `appuifw.ListBox` object and preferences and event descriptions are shown using an `appuifw.Form` object (Figure 9.2). The core function is `update_list()`, which, not surprisingly, updates the event list. The list is retrieved according to the user's preferences which are handled by the `show_preferences()`, `save_preferences()` and `load_preferences()` functions. If the user selects an event

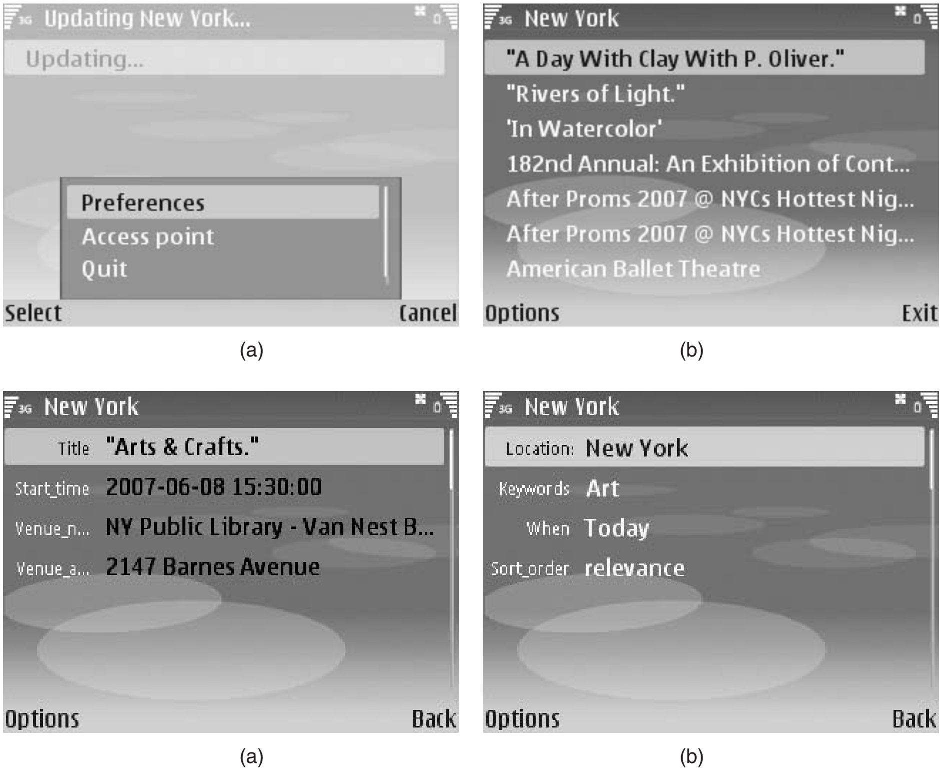


Figure 9.2 EventFu (a) menu, (b) event list, (c) event form and (d) preferences dialog

in the list, an event description form is opened (show_event()). In this form, the user may choose to see the full event description (show_description()) that is returned in HTML format from Eventful.

9.3.1 Constants and Preferences Form

Example 90 presents the first part of EventFu. First, you should replace the application key constant APP_KEY with the personal key you received from Eventful. The search requests are made to the address specified in SEARCH_URL. The next three constants are related to configuration: CONF_FILE gives a file name for the local database that keeps the user's preferences, DESCRIPTION_FILE specifies a file in which event description text is saved temporarily for showing; UPDATE_INTERVAL specifies the interval in seconds between event list updates. Note that a short interval may incur higher bandwidth usage and cost, depending on your data plan and it drains the battery faster.

Constant lists WHEN and ORDER correspond to parameters when and sort_order which are part of the service request. The lists contain all possible values for the parameters. The user may choose values in the preferences form that presents these lists in combo fields, as we see soon.

Example 90: EventFu (1/5)

```
import appuifw, e32, urllib, socket, e32dbm, json, os.path, os

APP_KEY = "5IsN4V9AdLwI3Dde"
SEARCH_URL = "http://api.evdb.com/json/events/search?"

CONF_FILE = u"E:\\Data\\Eventfu\\eventfu.cfg"
DESCRIPTION_FILE = u"E:\\Data\\Eventfu\\eventfu.html"
UPDATE_INTERVAL = 600

if not os.path.exists("E:\\Data\\Eventfu"):
    os.makedirs("E:\\Data\\Eventfu")

WHEN = [u"All", u"Future", u"Past", u"Today",
        u"Last week", u"This Week", u"Next Week"]
ORDER = [u"relevance", u"date", u"title",
         u"venue_name", u"distance"]
EVENT_FIELDS = [u"title", u"start_time", u"venue_name",
                u"venue_address"]

def show_prefs():
    if appuifw.app.title.find("Updating") != -1:
        return
    form = appuifw.Form([
            (u"Location", "text", prefs.get("Location", u"")),
            (u"Keywords", "text", prefs.get("Keywords", u"")),
            (u"When", "combo", (WHEN, 3)),
            (u"Sort_order", "combo", (ORDER, 0))],
            appuifw.FFormEditModeOnly)
    form.menu = []
```

```
    form.save_hook = save_prefs
    form.execute()
```

`EVENT_FIELDS` lists the event attributes that are included in the event form. The list is a subset of all possible attributes that the Eventful service records for an event. The full list, available at ***http://api.eventful.com/docs/events/search***, contains over 20 attributes, so showing all of them to the user would be rather impractical. You may change the list to contain the attributes that are most interesting to you.

Figure 9.2(d) shows the preferences form which is constructed by the function `show_prefs`. In this form, the user can specify the city or other location where the returned events should take place. To restrict the listing further, she may give a keyword that describes the event. She may also specify when the event should take place, the choices being listed in the list `WHEN`. The matching events are ordered according to one of the attributes in `ORDER` which is specified in the last field.

First, the function `show_prefs()` makes sure that a previous search request is not in progress, as we do not want the requests to queue up. After this, an editable form is constructed. When the user closes the form or saves it explicitly with the Save option in the menu, the function `save_prefs()` is called, as specified in the `form.save_hook` variable. Finally, `form.execute()` makes the form visible.

9.3.2 Storing Preferences

Choices in the preference form are saved in a local database, so the user does not have to type them in every time she starts the application. Local databases were first introduced in Section 6.3. They look and behave like an ordinary dictionary. However, in contrast to dictionaries, which are destroyed when the application closes, local databases are backed up to a file.

Example 91 shows the function `save_prefs()` that saves the current preferences. The function is given a dictionary `new_prefs` that contains the values from the preferences form. A new database is opened in file `CONF_FILE`. Attributes `When` and `Sort_order` require special treatment since they contain a list of values and the index of the chosen item in a tuple. In this case, we replace the tuple with the chosen value.

Example 91: EventFu (2/5)

```
def save_prefs(new_prefs):
    db = e32dbm.open(CONF_FILE, "nf")
    for label, type, value in new_prefs:
        if label == "When" or label == "Sort_order":
            value = value[0][value[1]]
        prefs[label] = value
```

```
        db[label] = value.encode("utf-8")
    db.close()
    timer.cancel()
    timer.after(0, update_list)
    return True

def load_prefs():
    global prefs
    try:
        prefs = {}
        db = e32dbm.open(CONF_FILE, "r")
        for k, v in db.iteritems():
            prefs[k] = v.decode("utf-8")
        db.close()
    except Exception, x:
        prefs = {}
    return prefs
```

Once the preferences have been stored in the database, a list update is triggered. This is done by the `timer` object which is introduced with the function `update_list()`. The function returns `True` to approve the changes made to the form.

The function `load_prefs()` performs the reverse operation: it opens the database and copies values from it to a dictionary, `prefs`, that keeps the current preferences. Strings, which are stored in the UTF-8 format in the database, are decoded back to Unicode strings. If anything goes wrong in loading (for example, if the database does not exist), an empty set of preferences is returned.

9.3.3 Updating Events using the Eventful API

The Eventful Web API is used to retrieve events according to the user's preferences. Titles of the retrieved events are presented in a list, like the one that is shown in Figure 9.2(b). The list is constructed by the function `update_list()` that is presented in Example 92.

Example 92: EventFu (3/5)

```
def update_list():
    global alive, events
    lprefs = {'app_key': APP_KEY, 'page_size': '10'}
    for k, v in prefs.items():
        if v:
            lprefs[k.lower()] = v

    listbox.set_list([u"Updating..."])
    appuifw.app.title = u"Updating %s..." % prefs.get('Location', u"")
    try:
        url = SEARCH_URL + urllib.urlencode(lprefs)
        res = urllib.urlopen(url).read()
        events = json.read(res)['events']['event']
        titles = []
```

```
        for event in events:
            titles.append(unicode(event['title']))
        listbox.set_list(titles)
        appuifw.app.title = prefs['Location']
    except:
        listbox.set_list([u"Could not fetch events"])
        appuifw.app.title = u"EventFu"
    if alive:
        timer.after(UPDATE_INTERVAL, update_list)
```

Fields of the preferences form map directly to parameters of the corresponding search request. Parameters can be given in the requested URL, so again we can use a standard `urllib.urlopen()` call to access the service. The dictionary `lprefs` is filled with the parameters: `app_key` specifies your personal Eventful application key and `page_size` specifies the number of events returned. The other parameters are filled in by looping over the preferences dictionary, `prefs`. Keys are converted to lower case and empty values are omitted.

Before a call is made to Eventful, the application title is changed to notify the user about the ongoing update. Since the application is single-threaded, that is, it performs only one operation at time, a call to Eventful makes the application unresponsive for a short period. Thus, it is important that we inform the user that the application has not crashed. By making the application multi-threaded, and somewhat more complex, we could avoid the pause.

The actual call to Eventful and parsing of the results are enclosed in a `try–except` block. If the network is unavailable, as often happens with a mobile device, or the Eventful service returns unknown or erroneous results, we want to inform the user properly. Since the application makes the same request again after some time to receive new events, a temporary network failure is not a fatal error and the application may continue working without the user even noticing the glitch.

The service returns the matching event list encoded in JSON, which was first introduced in Section 8.3. A JSON parser translates a JSON-encoded message to the corresponding data structures automatically. The structure of Eventful's replies are fully documented at ***http://api.eventful.com***. Here we are only interested in the list of events that match the user's preferences, which is assigned to the variable `events`. From this list, we extract the event titles to the list `titles` which is then updated to the list box.

To receive new events to the list automatically, some mechanism must be used to call the function `update_list()` every once in a while. We can employ here the same solution as in the GSM location application in Section 6.4, namely the `e32.Ao_timer` object. After the function has finished with updating, we set up the timer to call the same function, `update_list()`, again after `UPDATE_INTERVAL` seconds unless the user has asked the application to quit and the variable `alive` is set to `False`.

If the user has changed some values in the preferences form, the list must be updated immediately – we cannot expect the user to wait for, say, ten minutes to see the effect of the new settings. Because of this, the function `save_prefs()` cancels the current timer and sets up a new one that calls `update_list()` immediately. The benefit of using the timer object to trigger the update is that `save_prefs()` may return immediately and it does not have to wait for `update_list()` to finish, as would be the case with a normal function call.

9.3.4 Event Form and Event Description

Figure 9.2(c) shows the event form which is constructed by the function `show_event()` (see Example 93). The form shows a subset, specified in `EVENT_FIELDS`, of event attributes that are returned by Eventful. Usually we hard-code the structure of UI elements in our applications but, as shown by this example, we can construct the elements on the fly as well as based on some external data. First, however, we make the attribute names more readable by converting their first characters to upper case and their values to Unicode, as Eventful returns them in the UTF-8 format.

Example 93: EventFu (4/5)

```
def show_description():
    global desc
    f = file(DESCRIPTION_FILE, "w")
    f.write(u"<html><body>%s</body></html>" % desc)
    f.close()
    lock = e32.Ao_lock()
    viewer = appuifw.Content_handler(lock.signal)
    viewer.open(DESCRIPTION_FILE)
    lock.wait()

def show_event():
    global desc
    if not events:
        return

    event = events[listbox.current()]
    form_elements = []
    for field in EVENT_FIELDS:
        if field in event and event[field]:
            key = field.capitalize()
            value = event[field].decode("utf-8")
            form_elements.append((key, "text", value))

    form = appuifw.Form(form_elements, appuifw.FFormViewModeOnly)
    if 'description' in event:
        desc = event['description'].decode("utf-8")
        form.menu = [(u"description", show_description)]

    form.execute()
```

Besides attributes that contain only a word or two, such as the event location and time, Eventful returns also a free-form description of the event, which may be arbitrarily long. The description could hardly fit in the event form. Instead, we use the phone's default browser to show the description. The user may choose to see the description from the form menu which calls the function `show_description()`.

The function `show_description()` works in a straightforward manner: The description, which is embedded in an HTML page, is written to a temporary file, `DESCRIPTION_FILE`. The `appuifw.Content_handler` object is used to open the file. The content handler infers the file type and opens an appropriate viewer. A similar pattern was used in Example 67 that showed a downloaded image using the standard image viewer.

9.3.5 Access Point Dialog and User Interface Functions

Example 94 presents the already familiar UI functions for EventFu. The function `access_point()`, which is triggered by the corresponding menu item, is used to select the default network connection or access point for the application. If no default access point is chosen, the periodical list update may cause the access point selection dialog to pop up again and again. A detailed description of the access point functions can be found in Chapter 8.

Example 94: EventFu (5/5)

```
def access_point():
    ap_id = socket.select_access_point()
    apo = socket.access_point(ap_id)
    socket.set_default_access_point(apo)

def quit():
    global alive
    alive = False
    timer.cancel()
    app_lock.signal()

events = None
alive = True
timer = e32.Ao_timer()
appuifw.app.exit_key_handler = quit
appuifw.app.title = u"EventFu"
appuifw.app.menu = [(u"Preferences", show_prefs),
                    (u"Access point", access_point),
                    (u"Quit", quit)]

appuifw.app.body = listbox = appuifw.Listbox([u""], show_event)

load_prefs()
update_list()
app_lock = e32.Ao_lock()
app_lock.wait()
```

When the application starts, we load preferences (`load_prefs()`) and update the event list (`update_list()`). When you run the application for the first time, the update fails because of missing preferences. This is fixed by filling in the preferences form.

Note that we need to cancel the `timer` object when the application is about to exit. A timer that is left active after the application has quit causes the PyS60 interpreter to crash eventually. By setting `alive = False`, we make sure that the `timer` is not activated again by the `update_list()` function after it has been cancelled.

The current version of Eventfu uses only one function in the Eventful Web API. Given that the API contains over 100 functions in total, it should not be hard to find ways to extend the application. Consider also combining EventFu with the next application, InstaFlickr. For example, with this combination you could automatically tag any photos taken, say, in a pony exhibition with appropriate attributes fetched from Eventful before sending them to Flickr!

9.4 InstaFlickr: Shoot and Upload Photos to Flickr

With InstaFlickr, it takes only about 10 seconds to shoot a photo and upload it to your Flickr account (the actual time depends on your network connection). No need to configure or type anything – just point, shoot and upload! This application is made possible by the Flickr Web API (***www.flickr.com/services/api***) which allows you to upload photos to your account programmatically.

There are two tricky issues in the implementation of this application. First, as the photos are uploaded to a personal Flickr account, we must go through Flickr's four-step authentication process. Second, since this application sends a lot of data, in contrast to MopyMaps! and EventFu,



(a) (b) (c)

Figure 9.3 InstaFlickr (a) menu, (b) viewfinder and (c) upload progress bar

which mostly receive it, we have to use a special method for making the requests. Luckily, we can solve these tasks using techniques that have been introduced already.

The InstaFlickr source code is divided into six parts. As before, the parts should be combined into one file to form the full application. The parts cover the following functionalities:

- constants and result parsing
- handling Flickr tokens
- making signed calls to Flickr
- data uploading
- taking photos and implementing a progress bar
- user interface functions.

You need to apply for a Flickr API key to use the application. You can get one at **www.flickr.com/services/api/keys/apply**. Apart from the uploading functionality, the application is similar to the previous examples that use the camera, such as Example 36. The user may start the viewfinder from a menu item (`start_viewfinder()`) and take a photo with the select key (`take_photo()`). The photo is then sent automatically (`flickr_signed_call()`) to a previously defined (`new_token()`) Flickr account.

9.4.1 Constants and Result Parsing

Example 95 defines the constants and the `naive_xml_parser()` function that was first introduced in MopyMaps! Example 87. In a similar way to the Yahoo! Map API, functions in the Flickr API return a small XML message that can be parsed using this simple approach.

Example 95: InstaFlickr (1/6)

```
import e32, md5, urllib, httplib, camera, key_codes, os, os.path

API_KEY = "c62dcNiOgTeVhAaLdIxDsdf17ee3afe9"
SECRET = "c932fFxAkK9E648d"
API_URL = "http://flickr.com/services/rest/"
UPLOAD_URL = "http://api.flickr.com/services/upload/"

IMG_FILE = u"E:\\Data\\Instaflickr\\instaflickr.jpg"
TOKEN_FILE = u"E:\\Data\\Instaflickr\\instaflickr.txt"
BLOCKSIZE = 8192

PROGRESS_COLOR = (0, 255, 0)
TEXT_COLOR = (255, 0, 0)
```

```
if not os.path.exists("E:\\Data\\Instaflickr"):
    os.makedirs("E:\\Data\\Instaflickr")

def naive_xml_parser(key, xml):
    key = key.lower()
    for tag in xml.split("<"):
        tokens = tag.split()
        if tokens and tokens[0].lower().startswith(key):
            return tag.split(">")[1].strip()
    return None
```

The Flickr API key consists of two parts: the public application key and its secret counterpart. Once you have applied for your personal key, you can see it and the corresponding secret key at ***www.flickr.com/services/api/keys***. Replace `API_KEY` in the code with your personal key and `SECRET` with the secret key that corresponds to it. The constants `API_URL` and `UPLOAD_URL` contain the base address of Flickr API functions and the photo upload function, correspondingly.

`TOKEN_FILE` specifies a file which is used to save a token that gives access to a chosen Flickr account. Given this token, `API_KEY` and `SECRET`, it is possible to modify the Flickr account of the person that approved the token originally. Thus, you should keep these strings private. `IMG_FILE` is used to save the image in a file temporarily.

`BLOCK_SIZE` specifies how many bytes of image data is sent at a time to Flickr. The progress bar is updated after each block has been sent. However, there should be no need to change this value. `PROGRESS_COLOR` defines the color of the progress bar in RGB and `TEXT_COLOR` defines the color of the text, unsurprisingly.

9.4.2 Handling Flickr Tokens

To upload photos to a Flickr account, we need permission from the account holder. However, the account holder should not trust her username and password to our application – or to any third-party application whatsoever. Instead, she can request a token from Flickr that is specifically assigned to a certain application and which she can reject at any time.

This is how the token is generated. The developer must perform the following steps:

1. Request an application key from Flickr (if you have not done that already).
2. Go to the 'Your API keys' page at ***www.flickr.com/services/api/keys***.
3. Choose 'Edit key details'.
4. For 'Authentication Type' choose 'Mobile Application'.

5. For 'Mobile permissions' choose 'Write'.
6. Write down the newly generated 'Your authentication URL'.

Now, the person to whose account the photos are to be uploaded must perform the following steps:

1. Go to the authentication URL that was shown in Step 6, above.
2. Log in to your account and select the button 'Ok, I'll allow it' when you are asked about an application wanting to link to your account.
3. Write down the nine-digit code that is shown.
4. Open InstaFlickr.
5. Choose 'New Token' menu item.
6. Type in the nine-digit code.

If the token is accepted, InstaFlickr says 'Token saved!'. If InstaFlickr could not save the token, you should double-check that `API_KEY`, `SECRET` and the nine-digit code have been typed in correctly. Note that the nine-digit code can be used only once. If you request a new code, the previous code becomes invalid. If you want to change the account to which the photos are uploaded, the owner of the new account must request a new nine-digit code from the authentication URL and input it to the 'New Token' dialog in InstaFlickr.

You can find detailed instructions about the authentication process at the Flickr services website. At time of printing, the how-to for mobile applications can be found at ***www.flickr.com/services/api/auth.howto.mobile.html***.

Example 96 shows how the tokens are handled internally. The function `new_token()` is called when the user chooses the menu item 'New Token'. The token given by the user is assigned to the variable `mini_token`. Since there is no guarantee that this is actually the application for which the nine-digit code was granted by the user, we must certify our identity to Flickr with the nine-digit code.

Example 96: InstaFlickr (2/6)

```
def load_token():
    global flickr_token
    try:
        flickr_token = file(TOKEN_FILE).read()
    except:
        new_token()

def new_token():
    global flickr_token
```

```
    mini_token = appuifw.query(u"Give Flickr mini-token " +
            "(e.g. 123456100)", "number")
    if not mini_token:
        return

    params = {"method": "flickr.auth.getFullToken",
              "api_key": API_KEY,
              "mini_token": str(mini_token)}
    flickr_token = naive_xml_parser("token", flickr_signed_call(params))
    if flickr_token:
        try:
            f = file(TOKEN_FILE, "w")
            f.write(flickr_token)
            f.close()
            appuifw.note(u"Token saved!", "info")
        except:
            appuifw.note(u"Could not save token", "error")
    else:
        appuifw.note(u"Invalid token", "error")
```

This is done by sending a request to a Flickr API function called `flickr.auth.getFullToken()`. As in the previous applications, we fill in the request parameters in a dictionary `params`. However, this time we do not make the call directly but by a special function `flickr_signed_call()` that *cryptographically signs* the request. This sounds fancy, but actually it is just a clever, simple trick that is described in Section 9.4.3.

If the code is accepted, Flickr replies with the following type of XML message:

```
<auth>
  <token>433445-76598454353455</token>
  <perms>write</perms>
  <user nsid="11223344556@N01" username="Mopy" fullname="N.N"/>
</auth>
```

Here the tag `token` contains the final token that gives access to the user's Flickr account. We can extract this value with the `naive_xml_parser` as described in the MopyMaps! section. We save this string to the file `TOKEN_FILE`. Next time the user opens InstaFlickr, the token is read from the file by the function `load_token()`. This way, the user has to authenticate to the system only once; after the first time, uploading is really instantaneous. The `load_token()` function presents the 'New token' dialog automatically if a `TOKEN_FILE` was not found.

9.4.3 Making Signed Calls to Flickr

The function `flickr_signed_call()` in Example 97 makes a signed call to Flickr using the given parameters `params`. Why does the request have to be signed in the first place? The previous applications worked fine without any signing.

Example 97: InstaFlickr (3/6)

```
def flickr_signed_call(params):
    keys = params.keys()
    keys.sort()
    msg = SECRET
    for k in keys:
        if k != "photo":
            msg += k + params[k]

    params['api_sig'] = md5.new(msg).hexdigest()
    if "photo" in params:
        return flickr_multipart_post(params)
    else:
        url = API_URL + "?" + urllib.urlencode(params)
        return urllib.urlopen(url).read()
```

In contrast to MopyMaps! and EventFu, InstaFlickr deals with someone's personal data. It has the power to delete all the photos in the user's Flickr account. In contrast, the previous applications made read-only requests to public data. Especially if your phone is connected to the Internet by an insecure wireless LAN, it is easy for anyone to eavesdrop the requests that are sent out by the application. If the requests were not signed, the eavesdropper could then pretend to be you and send modified requests to Flickr with disastrous effects.

Signing makes these man-in-the-middle attacks practically impossible. Signing is made effective by the `SECRET` string that is known only by your application and the recipient, in this case Flickr. Since `SECRET` is never sent anywhere, you can securely communicate with the other party who knows the `SECRET`, as long as you keep the phone holding the `SECRET` safe.

Signing works as follows: the parameters of the request, `params`, except for any image data, are combined into a single string `msg` that begins with the `SECRET` string. The order of the parameters is important so that the recipient can re-construct the same message, thus, the parameters are sorted in alphabetical order. Then we use a well-known *cryptographic hash function* called MD5 to compute a long number based on the newly formed string `msg`.

MD5 is designed to generate a unique result so that no other input string produces exactly the same number or signature. For brevity, the number is saved in hexadecimal notation. The signature is assigned to the request key `'api_sig'` in addition to the other parameters, `params`, as specified in the Flickr API. This process is described in the Flickr authentication specification at ***www.flickr.com/services/api/auth.spec.html***.

The result of this signing procedure is that anyone who knows the `SECRET` string can easily reproduce `msg` based on the request parameters. She can then compute the corresponding signature and check that it

matches with `'api_sig'`. On the other hand, someone who does not know `SECRET` cannot change the parameters without the recipient noticing, as she is unable to produce the correct signature.

After the signature has been computed, `flickr_signed_call()` chooses one of the two ways to make the actual request. If the parameters contain the key `'photo'`, a special function `flickr_multipart_post()` is chosen that is suitable for large requests – in this case the request contains the data of a full-scale photo. Otherwise, the request is made with the `urllib.urlopen()` function in a similar way to the previous example applications.

9.4.4 Data Uploading

Example 98 contains the function `flickr_multipart_post()` that implements a standard HTTP request method called multipart POST. This is the standard way to upload files to the web. Unfortunately, however, this method is not supported directly by Python's standard web modules, so we must provide an implementation of our own.

Example 98: InstaFlickr (4/6)

```
def flickr_multipart_post(params):
    BOUNDARY = "----ds9io349sfdfd!%#!dskm"
    body = []
    for k, v in params.items():
        body.append("--" + BOUNDARY)
        part_head = 'Content-Disposition: form-data; name="%s"' % k
        if k == "photo":
            body.append(part_head + ';filename="up.jpg"')
            body.append('Content-Type: image/jpeg')
        else:
            body.append(part_head)
        body.append('')
        body.append(v)

    body.append("--" + BOUNDARY + "--")
    body_txt = "\r\n".join(body)

    proto, tmp, host, path = UPLOAD_URL.split('/', 3)

    h = httplib.HTTP(host)
    h.putrequest('POST', "/%s" % path)
    h.putheader('content-type',
                "multipart/form-data; boundary=%s" % BOUNDARY)
    h.putheader('content-length', str(len(body_txt)))
    h.endheaders()

    try:
        offs = 0
        for i in range(0, len(body_txt), BLOCKSIZE):
            offs += BLOCKSIZE
            h.send(body_txt[i: offs])
```

```
            progress_bar(min(1.0, offs / float(len(body_txt))))

        errcode, errmsg, headers = h.getreply()
        return h.file.read()
    except:
        return None
```

The implementation, which includes the lines from the beginning of the function to the `try` statement, requires some understanding of the HTTP protocol. There is nothing particularly difficult in it and you do not need to understand it to use PyS60 successfully. Because of this, we omit the explanation here. Anyone interested in the protocol can find several comprehensive explanations on the web by searching for 'multipart post'.

However, this implementation contains one interesting detail: Once the message has been constructed in the string `body_txt`, it is not sent to the destination URL in one part. Since the image data can take 50–100 kilobytes, sending the data over any wireless network will take some time. Because of this, we send the data in parts, block by block. After each block is sent, a progress bar is updated to depict the progress of the transfer. Figure 9.3(c) shows a progress bar in action.

Piecewise sending is implemented by the `for` loop inside the `try–except` block at the end of `flickr_multipart_post()`. The loop variable `i` goes from 0 to the message size, increasing the variable by `BLOCKSIZE` after each iteration. During each iteration, a block of data starting at `i` is sent to the recipient. After that, the function `progress_bar` is called with a single parameter that denotes the progress in the scale from 0 to 1.0. Implementation of `progress_bar()` is described in Section 9.4.5.

After all the data has been sent successfully, `flickr_multipart_post()` reads a reply from the recipient and returns it to the caller. If any errors occur during sending, `None` is returned instead.

9.4.5 Taking Photos and the Progress Bar

Example 99 implements the camera functionality, the progress bar and a function that draws status messages on screen. Functions related to the camera, `finder_cb()`, `start_viewfinder()` and `take_photo()` are based on Example 36 in Section 5.4.

Example 99: InstaFlickr (5/6)

```
def progress_bar(p):
    y = canvas.size[1] / 2
    max_w = canvas.size[0] - 30
    canvas.rectangle((15, y, p * max_w, y + 10), fill = PROGRESS_COLOR)
```

```
def show_text(txt):
    s = canvas.size
    canvas.text((10, s[1] / 2 - 15), txt, fill=TEXT_COLOR, font="title")

def finder_cb(im):
    canvas.blit(im)

def start_viewfinder():
    if flickr_token:
        camera.start_finder(finder_cb)
        canvas.bind(key_codes.EKeySelect, take_photo)
    else:
        appuifw.note(u"Give a Flickr token first", "error")

def take_photo():
    canvas.bind(key_codes.EKeySelect, None)
    camera.stop_finder()
    show_text(u"Hold still!")
    image = camera.take_photo(size = (640, 480))
    s = canvas.size
    canvas.blit(image,target=(0,0,s[0],(s[0]/4*3)), scale=1)
    show_text(u"Uploading to Flickr...")
    image.save(IMG_FILE)
    jpeg = file(IMG_FILE, "r").read()
    params = {'api_key': API_KEY,
              'title': 'InstaFlickr',
              'auth_token': flickr_token,
              'photo': jpeg}
    ret = flickr_signed_call(params)
    canvas.clear((255, 255, 255))
    if ret:
        show_text(u"Photo sent ok!")
    else:
        show_text(u"Network error")
```

After a photo has been shot, the function `take_photo()` loads the image data from the image file `IMG_FILE` and prepares an upload request in `params` to be sent to Flickr. The request contains both the application key `API_KEY` that identifies this particular application and the token `flickr_token` that identifies the Flickr account which should receive the photo.

At this point, the user could assign a title or tags to the photo. In this example, the title is fixed to 'InstaFlickr'. See the Flickr API documentation at ***www.flickr.com/services/api/upload.api.html*** for possible attributes that can be attached to the photo.

The function `progress_bar` is called whenever a block of data has been sent. The parameter `p` is a floating-point value between 0 and 1.0 that denotes the progress. Based on this value, we draw a rectangle in the middle of canvas that grows from left to right as the data is sent. The effect can be seen in Figure 9.3(c).

The function `show_text()` is used to draw status messages in the middle of canvas. The function should be rather self-explanatory.

9.4.6 User Interface Functions

Example 100 explains the remaining functions related to the user interface. The function `access_point()` is familiar from the previous application, EventFu. It is described with Example 94.

Example 100: InstaFlickr (6/6)

```
def access_point():
    ap_id = socket.select_access_point()
    apo = socket.access_point(ap_id)
    socket.set_default_access_point(apo)

def quit():
    camera.stop_finder()
    app_lock.signal()

appuifw.app.exit_key_handler = quit
appuifw.app.title = u"InstaFlickr"
appuifw.app.menu = [(u"Take photo", start_viewfinder),
                    (u"New token", new_token),
                    (u"Access point", access_point),
                    (u"Quit", quit)]

appuifw.app.body = canvas = appuifw.Canvas()
canvas.clear((255, 255, 255))
show_text(u"Welcome to InstaFlickr")

load_token()
app_lock = e32.Ao_lock()
app_lock.wait()
```

Note that the Flickr token is loaded with the function `load_token()` when the application starts. If no token has been defined previously, the function shows a 'New token' dialog.

You may find inspiration for your own extensions in the Flickr mobile tools page at ***www.flickr.com/tools/mobile***. For example, InstaFlickr could be combined with the GSM location example (Example 50) so that the photos could be tagged with the current location – see Zonetag at ***http://zonetag.research.yahoo.com*** for a reference.

9.5 Summary

In this chapter, we have presented three working applications, MopyMaps!, EventFu and InstaFlickr, which are based on three different web services. These applications delegate most of the difficult work to an

external service, which results in remarkably light processing on the client side. For example, consider MopyMaps!, which implements a working global map client in around 100 lines of code! With modern web services and a bit of imagination you can build amazingly functional applications with unbelievably few lines of code.

10

Effective Python for S60

In the previous chapters we have deliberately avoided some advanced programming techniques to focus on the functionalities of the mobile platform. Many of the examples could be made shorter and more elegant with techniques that are in this chapter.

Section 10.1 presents some powerful constructs of the Python language that we omitted in the earlier chapters. In Section 10.2, we describe how you can avoid clumsy static constructs by deciding what to do with run-time introspection.

Techniques presented in Section 10.3 really put the word rapid into rapid development. These methods, related to modularization, extensibility and updating of PyS60 programs, have proven to be immensely useful not only during the development process, but also for deployment and maintenance.

Section 10.4 shows another viewpoint of the examples in this book. We summarize notable recurring patterns in the structure of PyS60 programs, which help you to generalize the examples of this book to practically any real-world setting. Finally, in Section 10.5, we present seven crystallized thoughts about PyS60 programming.

All in all, be prepared for a serious productivity boost!

10.1 Powerful Language Constructs

Python is a very expressive programming language. You can perform complex operations with just a few lines of code. In many cases, compact expressions translate to easy readability and efficient processing, thus these forms are often preferred over lengthier ones. Luckily, Python invites you to write clean and simple code once you know the nuances of the language. As a matter of fact, the following language constructs are the bread and butter of elegant Python.

10.1.1 List Comprehensions

As seen in the previous chapters, many tasks boil down to various operations on lists. Python provides a compact syntax for constructing new lists, called list comprehensions. Using list comprehensions you can construct a new list either from scratch or based on a previous sequence of objects.

Using a simple loop, you can construct a list of ten user names as follows

```
a = []
for x in range(10):
    a.append("user%d" % x)
```

However, you can get the same result with a list comprehension using only one line of code (as shown in Example 101).

Example 101: List comprehension

```
a = ["user%d" % x for x in range(10)]
```

Not only is this expression shorter, but it is also more efficient since we don't have to call the `append()` function ten times. List comprehensions have the following syntax:

```
[<list item expression> for <iterator variable> in <list expression>
    (if <conditional>)]
```

In Example 101, the list item expression is `"user%d"% x`, the iterator variable is `x` and the list expression is `range(10)`. This example did not have any conditional element. No other elements are allowed in the list comprehension. The list item expression can be arbitrarily complex but it cannot contain statements such as `print`, `while` or `return`. However, it can contain another list comprehension, so you can create nested or multi-dimensional lists.

Recall the SMS search tool, Example 15, from Chapter 4. The example was similar to the following code:

```
import inbox, appuifw

box = inbox.Inbox()
query = appuifw.query(u"Type in the query", "text")

hits = []
for sms_id in box.sms_messages():
    msg = box.content(sms_id)
    if msg.find(query) != -1:
        hits.append(sms_id)
```

We can condense the example and make it more understandable using list comprehensions.

Example 102: SMS search using list comprehensions

```
import inbox, appuifw

box = inbox.Inbox()
query = appuifw.query(u"Type in the query", "text")

hits = [sms_id for sms_id in box.sms_messages()\
            if box.content(sms_id).find(query) != -1]
```

The above list comprehension becomes clear when you read it as follows: pick `sms_id` for each item of the list returned by `box.sms_messages()` where the return value of the function `box.content()` contains the string `query`.

This sentence captures the idea of the search tool much more comprehensively than a corresponding explanation for the original five-line version of the code.

You can solve many practical tasks using list comprehensions. For example, if you need to perform any operation for each element of a list and save the return values, you can use an expression such as the following:

```
results = [f(x) for x in my_list]
```

Python includes a built-in function called `map()` that produces the same result as the previous list comprehension as follows:

```
results = map(f, my_list)
```

Here, `f` is a callback function that is called for each item in the list `my_list`. For example, the following two lines produce an equivalent result: they convert a list of integers between 0 and 9 to a list of Unicode strings:

```
results = [unicode(x) for x in range(10)]
results = map(unicode, range(10))
```

Note that you do not have to use a list comprehension to construct a list whose elements are all equal. For example, you can initialize a list of `N` zero values simply as follows:

```
N = 10
zeros = [0] * N
```

You can choose all elements that satisfy a condition as follows

```
results = [x for x in my_list if x > 0]
```

Here you can replace `x > 0` with any valid `if` condition. For example, you can filter out all illegal characters in a string (Example 103).

Example 103: Input sanitization using list comprehensions

```
input = " this is, a TEST input!"
allowed = "abcdefghijklmnopqrstuvwxyz"
print "".join([c for c in input if c in allowed])
```

The output is `thisisainput`. The trick here is that the loop can contain any iterable object, in this case a string, that loops over all characters in it when treated as a sequence.

10.1.2 Dictionary Constructor

Besides lists, another data type that can be found in almost every Python program is a dictionary. The most typical use for the dictionary is to get a value, given the corresponding key, in an efficient manner. However, sometimes you can just use the fact that the dictionary cannot contain duplicate keys.

For example, you can count the number of distinct items in a list as follows:

```
a = [1, 2, "a", 1, "b", 2, "b"]
d = {}
for x in a:
    d[x] = True
print len(d)
```

The output is 4 in this case. As with the lists above, it may feel unnecessarily verbose to use four lines of code to say 'Construct a dictionary `d` whose keys are the items of `a`'.

Luckily, the built-in `dict()` function saves us. The function takes a list of key–value pairs as tuples and returns a new dictionary based on the list. We can re-write the previous example as Example 104.

Example 104: Dictionary constructor

```
a = [1, 2, "a", 1, "b", 2, "b"]
print len(dict([(x, True) for x in a]))
```

Here, we use a list comprehension to build a list of tuples based on the original list a. The new list is given to the dictionary constructor dict(), which returns a new dictionary whose size is then printed out.

In some cases you would like to retrieve a key given a value – the opposite operation of the normal dictionary usage. Since there may be multiple keys that have an equal value, you need to return a list of keys given a single value:

```
keys = [k for k, v in d.items() if v == x]
```

Here d is the dictionary and x is the value of interest. On the other hand, if only one key per value is enough and you need to perform the operation for multiple values, it makes sense to make a new dictionary by reversing the roles of the keys and values.

```
reverse_d = dict([(v, k) for k, v in d.items()])
key = reverse_d[x]
```

10.2 Introspection

One of the great strengths of a dynamic programming language, such as Python, is that it is possible to examine and even manipulate the program code at run-time. The idea of introspective programs may sound esoteric and mind-bending, but on examination it provides a clean way to solve some rather mundane tasks.

The core idea is simple: given that your program includes a function or an object named xyzzy, you can use the string "xyzzy" to access that object, rather than specifying the name in your source code. Thus, your program can choose an appropriate action to take based on a string originating from the network or from the user, without an if statement.

Introspection can be implemented in a straightforward manner using a *symbol table* that is maintained by the Python interpreter. The symbol table contains a mapping from all names defined in the program code to the corresponding objects. Actually, the symbol table can be thought of as a special dictionary that contains everything related to the internals of the current module. In some sense, it is like an internal address book for finding and executing code in a module.

At any point of execution, the interpreter maintains two separate symbol tables: one for the global scope and one for the local objects in the current function. Actually, the global keyword, which was first mentioned in Chapter 3, just instructs the interpreter to place an object in the global symbol table instead of the default local one.

In Python, functions are objects too and they can be found in the symbol table just like any other named entity in your program. Because of this, it is possible to retrieve and call a function by its name at run-time. With this approach, you can use input data to decide which functions to call on the fly, without having to hard-code all possible cases in your program beforehand.

In practice, this is the most important direct use of the symbol table in a normal application. Many other tricks on the symbol table are generally considered unnecessarily ugly and unsafe.

Let's start with a simple example that calls a function based on the user's input.

Example 105: Symbol table

```
import appuifw

def bark():
    appuifw.note(u"Ruff ruff!")

def sing():
    appuifw.note(u"Tralala")

func_name = appuifw.query(u"What shall I do?", "text")
func = globals()[func_name]
print globals()
func()
```

The program gets the string `func_name` from the user through a query dialog. This string is used as a key to the current global symbol table that is returned by the function `globals()`. If the key cannot be found in the symbol table, an exception is raised and the execution terminates. If the key is found, the corresponding function is called: try to type either 'bark' or 'sing' in the example above and see what happens.

After this, the symbol table is printed out to give you some idea what it actually contains. The exact output depends on what you typed in the dialog but it looks roughly like this:

```
{'__builtins__': <module '__builtin__' (built-in)>,
'name__': '__main__',
'bark': <function bark at 0xb7db1e64>,
'sing': <function sing at 0xb7db2e65>,
'func': <function bark at 0xb7db1e64>,
'func_name': 'bark',
'__doc__': None}
```

Here, the user told the program to bark, so from the symbol table you can see that the variable `func_name` is assigned the string `"bark"`. By default, the table contains some special entries such as `'__builtins__'`, which refers to the module containing Python's built-in

functions, '__name__', which is always '__main__' for the module that started the process and '__document__', which contains the documentation string for this module.

You can see that the functions bark() and sing() have entries in the table which point at the corresponding function objects. Since the user told us to bark, we retrieved the bark() function from the symbol table, using func_name as the key and assigned the retrieved function object to the variable func. We could have called the function bark directly without creating the new func variable, but it was included here to illustrate operation of the symbol table.

On the last line of Example 105, we finally call the function func. Depending on the input, either the bark() or sing() function is called. The main benefit of this approach is that we do not need a long list of if statements to decide what to do. Also, if we add a new function to the program, for instance yodel(), we could just implement the function without needing to change or add anything in the control logic. If a function corresponding to the string does not exist in the symbol table, an exception is thrown. Any real program should include a try–except clause to handle cases like this.

Using these ideas, we can re-write the phone-as-a-server example from Section 8.7. The original version has a separate if clause for every resource in the server. If we increase the number of resources shared by the server, the list of choices would soon become difficult to maintain. In contrast, we can use the global symbol table and call each function directly by the resource name.

Example 106: Introspective web service

```
import e32, json, camera, graphics, sysinfo, urllib

URL = "http://192.168.0.2:9000"
imei = sysinfo.imei()

def json_request(req):
    enc = json.write(req)
    return json.read(urllib.urlopen(URL, enc).read())

def RSC_screenshot_jpg():
    img = graphics.screenshot()
    img.save("c:\\python\\temp.jpg")
    data = file("c:\\python\\temp.jpg").read()
    return ("image/jpeg", data)

def RSC_battery():
    txt = "Current battery level is %d" % sysinfo.battery()
    return ("text/plain", txt)

def RSC_exit():
    global go_on
    go_on = False
```

```
go_on = True
msg = {}
while go_on:
    ret = {}
    for path in json_request(msg):
        rsc = "RSC_%s" % path[1:].replace(".", "_")
    if rsc in globals():
        ret[path] = globals()[rsc]()
    else:
        ret[path] = ("text/plain", "unknown resource")
    msg = ret
    e32.ao_sleep(5)
```

Here we use the same symbol table technique to call a function. However, we do not use the input string as such to define the function to call. Instead, we have prefixed the name of each function that defines a valid resource with `"RSC_"`. This is to ensure that the user cannot call any function in the program, such as the function `json_request()` that is meant only for internal use.

This is an extremely important security precaution. An untrusted outsider should be allowed to call only a restricted set of functions. If we used any user input to access the symbol table directly, as we did in the first example, the user could easily crash the program by accessing an unintended entry in the table. In the worst case, a malicious user might be able to destroy important data by controlling the program as she wishes. By prefixing all resource requests with `"RSC_"`, we make sure that the user can access only the functions that are serving valid resources.

10.3 Custom Modules and Automatic Updating

In this section, we explain how you can make modules by yourself for PyS60. The process in itself is almost trivial. However, custom modules are an extremely powerful feature of PyS60, with remarkably interesting implications. Not only can you structure your code more efficiently, but you can also make totally new kinds of programs, thanks to the dynamic nature of PyS60.

To demonstrate this, we show how to update your PyS60 applications automatically from the web and how to create a simple plug-in mechanism for your program.

10.3.1 Custom Modules

There is nothing really special in custom modules. You just create a source code file, put some custom functions in it and copy it to a certain location

on your mobile phone. After this, you can import the new module into your programs, in the same way as any standard PyS60 module.

Let's go through this process step by step. First, let's make a file that contains a simple function and the `import` statement that imports the modules needed by the function, as usual. Here, we use the `askword()` function from Example 10:

```
import appuifw

def askword():
    d = appuifw.query(u"Type your name", "text")
    appuifw.note(u"Hi " + str(word))
```

Save these five lines to a file called `mymodule.py` on your PC. Next, we need to upload this file to your phone. However, to be usable as a module, the file has to be copied to a special directory, `E:\Python\lib`, on your phone. If this directory does not exist, you have to create it first. This process is exactly the same as with the custom JSON module that we installed in Section 8.2.2.

After you have uploaded the file successfully to your phone, we can make another file that uses the custom module. Alternatively, you could test it using the Bluetooth console (see Appendix B). We can use Example 107 to test the new module.

Example 107: Importing a custom module

```
import mymodule, appuifw

appuifw.note(u"This is the main program")
mymodule.askword()
```

As you can see, we import the new module `mymodule` just like the standard module `appuifw`. As you could guess, the module's name is derived from the file name. After the module has been imported, you can use its functions in the usual way – just remember to prefix any function call with the module name, as in `mymodule.askword()` above.

When you execute the example, you first see a note saying 'This is the main program' and then you are asked to 'Type your name' when the execution proceeds to the `mymodule.askword()` function.

As you can see, making custom modules is really straightforward in PyS60. However, there is one important thing to remember: the `global` keyword that makes a variable accessible to many functions affects only one module. If you think about our explanation of the global symbol table in the previous section and how the keyword `global` relates to it, this outcome should feel logical, as each module has a symbol table of its own.

For instance, if you had declared the variable `word` global in `mymodule`, it would not have been visible to Example 107 automatically. This is beneficial, since you can treat each module as a separate unit independently from other modules.

Whenever you feel that your application gets too complex to handle in one file, separate it into several modules. Also, if you feel that a function would be useful for many applications, place it into a separate module so it can be imported to any application that needs it.

10.3.2 Extending Python for S60 in Symbian C++

It is also possible to extend PyS60 using Symbian C++. As a matter of fact, the standard PyS60 API is mostly implemented this way. Although this approach is much more complicated than making custom modules in Python, it enables you to connect directly to low-level services of Symbian OS.

For instance, if you want to use an accelerometer that may be built into your phone and it cannot be accessed using the standard PyS60 API, making a C++ extension is an appropriate choice.

To write a C++ extension for PyS60, you need:

- a PC with Windows
- C++ SDK for the S60 platform appropriate to your phone
- Python for S60 plug-in for the S60 C++ SDK

For details on how to write C++ extensions, see the PyS60 documentation and PyS60 wiki.

10.3.3 Automatic Updating

A file containing a PyS60 module, or any other source code in Python, is just an ordinary text file. Consequently, you can handle these source code files as any other files in Python. In particular, it is possible to download a source code file from the web and save it to `E:\Python\` or `E:\Python\Lib`, which makes it visible to the PyS60 interpreter.

This means that you can update your PyS60 programs using PyS60! Example 108 should clarify this.

Example 108: Updating PyS60 code from the web

```
import urllib

CODE = "mytest.py"
URL = "http://www.myownserver.com/pycode/"
```

```
code = urllib.urlopen(URL + CODE).read()
f = file(u"E:\\Python\\" + CODE, "w")
f.write(code)
f.close()
print "File %s updated successfully!" % CODE
```

As you can see, it uses the standard `urllib.urlopen()` function to download a file from the web, like many examples in Chapters 8 and 9. In this case, the file name `CODE` refers to a PyS60 source code file that we have made available to a web server at `URL`. Change the URL to point at your own web folder containing a file named `mytest.py`. The script downloads the file and saves it to the `E:\Python\` directory, so it can be found in the PyS60 interpreter's Run script menu. After running Example 108, a new file `mytest.py` should appear there as a new Python script.

It is impossible to overemphasize the usefulness of this little script. It has proven vital in projects carried out by the authors of this book in two respects. First, for some PyS60 developers, it can be the fastest way to upload code to the phone during development. If you are familiar with website development and you can edit files in the web easily, you can place and edit your PyS60 files on the web as well. When you need to test the code, you just execute this program and the source files are updated on your phone instantly. This is particularly convenient if you can use WiFi on your phone for network connection and you can run a local web server on your PC.

Second, this method has proved to be valuable in production settings as well. The authors of this book designed and implemented a large-scale urban game, called Manhattan Story Mashup, in New York in September 2006. More information about the game can be found in Section 11.2.

Because of many uncertainties, we were reluctant to freeze the game client a long time before the actual event. However, the players were given phones to play the game a week before the event, so we needed a method of updating the last-minute fixes to the players' phones.

Our solution is depicted in Figure 10.1. As you can see, the figure shows a PyS60 interpreter. However, we modified the application menu of the interpreter slightly. This is not difficult, since the PyS60 interpreter UI is implemented in Python (naturally!) and PyS60 is distributed as open source, so you are free to modify it in any way.

We added an Update StoryMashup item to the menu, as shown in the figure. This item executed a function similar to Example 108, which updated the game client code from the web.

Finally, just before the game started, we instructed our 160 players to update the latest version of the game client simply by selecting this item. As a result, all the players were using an identical version of the client, which included our last-minute changes. This method was a real life saver.



Figure 10.1 Automatic updating

10.3.4 Simple Plug-In Mechanism

We may take automatic updating even further. Not only can you download PyS60 modules from the web using Python, but you can also decide which modules to import at run time.

This makes your applications infinitely extensible. Depending on the user input, the physical environment or any other parameter, you can make your application request new functionality from the web. This leads to opportunities that come from science fiction, but, in simple terms, it allows you to easily make a plug-in mechanism for your applications.

Example 109: Plug-in mechanism

```
import urllib

URL = "http://www.myownserver.com/pycode/"

def download_plugin(plugin_name):
    filename = plugin_name + ".py"
    code = urllib.urlopen(URL + filename).read()
    f = file(u"E:\\Python\\Lib\\" + filename, "w")
    f.write(code)
    f.close()
    return __import__(plugin_name)

plugin_name = appuifw.query(u"Give plug-in name", "text")
print "Downloading plugin", plugin_name
plugin = download_plugin(plugin_name)
print "Plugin loaded!"
plugin.askword()
```

Example 109 wraps the download functionality of Example 108 in the function `download_plugin()`. As in the previous example, it downloads a PyS60 source code file from a specified URL on the web and saves it to a local directory. However, in this case, the file is saved to `E:\Python\Lib`, which makes the file able to be imported as a custom module, as we saw in Example 107.

The magic happens with Python's special `__import__()` function. Normally, when you import a module to your program, you have to specify the module's name in your source code, after the `import` statement. In contrast, the `__import__()` function lets you import a module at run time, by giving the module's name in a string. It returns the imported module as an object that you can use in the usual way.

In this example, the user can specify the plug-in name, `plugin_name`, that is loaded from the web. Let's assume that you type 'myplugin' in the dialog. Then, the `download_plugin()` function tries to download the file `http://www.myownserver.com/pycode/myplugin.py`.

You should make sure that such a file is available. When the file has been downloaded successfully, it is saved to `E:\Python\Lib`, after which it can be imported as any other module.

In this case, we import the module immediately using the `__import__()` function. The function `download_plugin()` returns the newly imported module in the variable `plugin`. We assume that the module contains a function called `askword()` that is then called in the last line of the example, which demonstrates that the new `plugin` module can be used just as usual.

Note that this example does not contain any precautions for exceptions. A real plug-in mechanism should make sure that the plug-in file is available and that it contains the necessary functions. This is easy to accomplish with `try-except` blocks, as described in Chapter 6.

10.4 Program Patterns

We have gone through over 100 code examples. The examples have demonstrated a wide array of topics from string handling, GUIs, MP3 players and 2D graphics to GSM locationing, AppleScript, JSON gateways and web services. From one point of view, this book could be considered a large grab bag of interesting things that one can do with a mobile phone.

On the other hand, one could claim that many examples differ merely on the surface level from others. You have probably noticed this phenomenon as well: many new modules and examples that we have introduced might have felt understandable to you at first sight. Even though the module name and, of course, its functionalities were different from what you had seen before, the new example often shared a similar structure with earlier examples.

This book is arranged according to subject areas, such as graphics, Bluetooth and network programming. Given that we believe that most of our readers are more interested in cool applications than theoretical computer science, this grouping felt appropriate.

However, we could have grouped the examples according to the *program pattern* that they follow. That is, instead of the subject area, we might categorize the programs according to how they are structured internally and how they interact with the outside world.

The following list presents one such categorization. For each pattern, we give a partial list of examples that follow this pattern. Note that a single program may be based on several interleaved patterns. For instance, the GSM cell ID mapper (Example 49) follows both the *updating* and the *application* patterns below.

- **Script:** these are small examples that execute sequentially from the beginning to the end and do not wait for external events to occur. Scripts are handy for automating small tasks. For instance, Examples 43, 63 and 79 follow this pattern, as well as all the examples in Chapter 3.
- **Application:** these are examples that are based on the S60 application user interface framework. The user may interact with the application, for example, by way of the application menu. Internally, the code relies on event callbacks. For instance, Examples 12, 49 and 73 and all the examples in Chapter 9 follow this pattern.
- **Updating:** this pattern is useful in cases where the program must update some data periodically. Typically, this is accomplished using the `e32.Ao_timer` object that is set to call a function at regular intervals. The GSM cell ID mapper in Example 49 and EventFu in Section 9.3 follow this pattern.
- **Event-driven:** external events can originate from some other source as well as the user interface. If the program must react to external events, typically by way of callbacks, we say that the program is event-driven. Examples include the game of Hangman in Section 4.5 that reacts to incoming SMS messages and the Instant Messenger in Section 8.6.2 that reacts to network events.
- **Game:** here, the crucial feature is an event loop that keeps the program active even without any external events. This pattern was thoroughly described in Section 5.5 and first exemplified by the drawing application in Example 33.
- **Client–server:** many examples in Chapters 7 and 8 are based on communication between a client and a server. The idea is that two independent programs communicate and thus affect each other's state.

The Bluetooth client in Example 59 and the server in Example 60 are illustrative of this pattern.

- **Concurrent:** in many real-world settings, a program must perform many things at the same time. The use of callback functions relies on the fact that some mechanism is listening to events in the background, although your program may simultaneously be busy doing something else. Thus, many examples we have seen actually work concurrently behind the scenes.

  In this book, we have deliberately avoided touching this issue too much, since it is notoriously difficult to program well-behaved concurrent programs. Fortunately, PyS60 often provides a way to avoid using threads and other concurrent programming techniques, such as active objects, used by Symbian OS, explicitly. However, in Section 8.6 we presented a message handler (Example 83) that shows a clean pattern of concurrency using threads.

Once you start building more complex applications of your own, seeing and using these patterns might prove useful. Before coding a single line of code, you can decide which of these patterns your application code should follow – often the choice is really evident. After this, you can browse through the examples that use that particular pattern. In the best case, you might be able to use one of the examples as a skeleton for your application and get past the 'empty editor' syndrome.

10.5 Summary

Python is a great language in which to write elegant and clean code. Writing clean code is the best way to avoid bugs in the first place and it gives you a feeling that you know what happens in your program at all times. Consequently finding any bugs is easier and fixes are simpler. Naturally the sense for elegant code is something that needs lots of hands-on practice and failures, to develop.

If you feel adventurous, you could take a look at the Python source code and find out how API calls are implemented. This is not as intimidating as it may first sound, since the code is typically understandable and, even if you do not understand it fully, it can point you in the right direction. Luckily PyS60 is open source so this is a real option!

There are often many different ways to accomplish one goal in programming. To get to your goal, you have to make a great number of small decisions. Often, alternatives are technically similar and the difference is mostly a matter of aesthetics. However, style does matter in programming. It makes sense to keep some rules of thumb in mind that may help you write better code.

It may be enlightening and entertaining to read the following short articles:

- 'Zen of Python' at ***www.python.org/dev/peps/pep-0020***
- 'Python Style Guide' at ***www.python.org/dev/peps/pep-0008***

These articles give deeper insights into issues of coding style in Python. Besides those articles, which discuss the Python language in general, we would like to present some thoughts that seem to be especially appropriate for PyS60:

- Make sure you understand it

  Don't write or use code which you do not fully understand. Debugging alien code is no fun. By keeping this principle in mind, you will become a master of PyS60 in no time.

- Don't copy and paste code

  Related to the previous advice, you should understand every line of code in your program. If you type in each line manually, you force yourself to think what you type.

- Be prepared for the real world

  Many bugs are caused by unexpected input values from the outside world. Try to keep the diversity of possible contexts in mind when designing your code. This is particularly important with mobile code, which is often used in many different environments. However, do not aim at universal solutions since they rarely exist. Also, remember to validate data before you use it and handle exceptions properly.

- Make it modular

  Complex programs can be kept simple if they are modularized properly. Separate your code logic into small functions and divide your code into modules.

- Keep it simple

  Use as few lines of code as possible, no fewer, no more. In Python, terseness is a virtue but not at the price of understandability.

- See patterns in your code and in that written by others

  Follow patterns that you have seen to work in practice. This way you can avoid debugging the same problems every time. Note, however, that patterns are just conceptual tools. Do not try to force your program to follow a pattern, if it doesn't feel natural. Also, do not use patterns as an excuse to copy and paste code.

- Be pragmatic

  As we said in Chapter 1, PyS60 is all about having your head in the clouds, your hands in mud and your feet on the ground. That is, come up with, implement and test working prototypes in rapid cycles. Only dinosaurs spend years thinking about grandiose software architectures – other reptiles have already evolved past that stage.

11

Combining Art and Engineering

This chapter brings together many of the concepts and techniques shown in the previous chapters. We provide a series of real-world application examples that combine art and engineering. They are all built and deployed using PyS60. Most of them have their origins in the field of digital art and are implemented by following the rapid prototyping approach with PyS60.

We hope to illustrate here that by having an application idea or a concept at the core of your actions, you can turn it into a fully working application by adopting and combining many of the previously introduced code examples. The applications we explain span from participatory and collaborative games to mobile multi-user applications controlling large displays, from interacting with a robot to physical computing using sensor boards and applications for controlling remote sound applications. Finally we show how you might turn your phone into a tool for creating mobile art.

Again we want to repeat here our message to all you creative and innovative people out there: use your talent, skills, ideas and energy to inspire the world! May this chapter and the entire book help you to do so!

11.1 MobiLenin

The MobiLenin system allows a group of people to interact simultaneously with a multi-track music video shown on a large public display using their personal mobile phones, effectively empowering the group with the joint authorship of the video. The system was implemented with a client–server architecture that includes server-driven, real-time control of the client UI written in PyS60 to guarantee ease of use. A lottery mechanism was built in as an incentive for interaction.

The MobiLenin system was a research project of one of the authors and his motivation as a music and new media artist and engineer was to create an interactive technology system that gives the audience the possibility of engaging in a new way in his live show – simply by interacting with the music and video on a large screen. The idea is to enhance people's concert experience by allowing them to interact with the artist in the virtual domain on the display.

11.1.1 System Architecture

The MobiLenin system comprises four components:

- a PyS60 client application running on a mobile phone
- an application running on a PC connected to the Internet
- an external server
- a large public display showing the music video.

There are several reasons why personal mobile phones are suitable user devices for this purpose. First, they are ubiquitous, as practically everyone has one. Second, they allow anonymous, wireless and mobile participation in a joint social and public group interaction. Third, the mobile phone provides a reliable return channel for delivering confidential user-specific information back to the user, such as a winning lottery coupon.

The PC application is implemented in Macromedia Director to count votes, operate the lottery mechanism, initiate the delivery of winning notifications and control the QuickTime player with its multi-track video as well as handling all the graphic elements on the public display and the sound. An external server component consisting of simple PHP scripts is placed on the Internet to act as a mediator between the public mobile data network and the PC running the main application. The communication between the two is done by HTTP. The external component also hosts the pictures for the lottery coupons to be fetched by the mobile devices upon initiation by the main application.

The large public display serves as the main interface for the user's interaction. Besides showing the music video, it indicates the start and end of a voting interval and the voting results and notifies the audience of somebody winning the lottery.

Figure 11.1 shows the state diagram of the system. When the voting interval starts, it is indicated in the client by an S60 popup note (Figure 11.2 (a)) and shown on the public display ('Vote now!'). The server opens the voting menu in each client, so that a vote is cast by selecting one of the given menu choices (Figure 11.2 (b)). If a vote is cast,

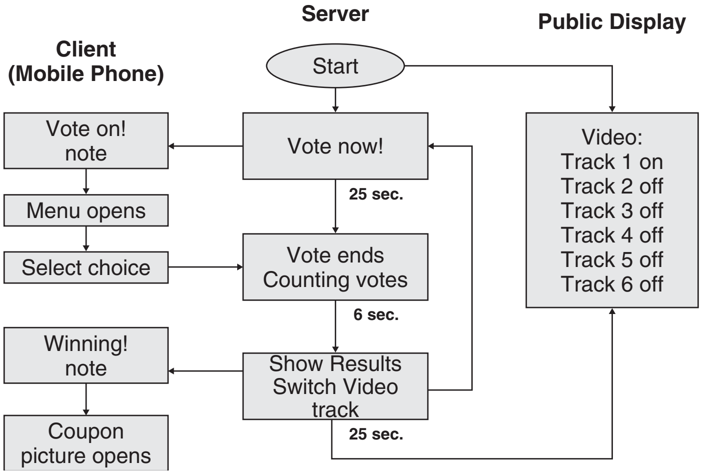


Figure 11.1 State diagram of the MobiLenin system

it is acknowledged by the client (Figure 11.2 (c)) and sent to the server. If the user wins in the lottery, a winning coupon is pushed to the client by the server and presented to the user (Figure 11.2 (d)).

After a 25-second voting interval is over, 'Counting votes!' is shown on the public display. The server counts the votes and, after six seconds, the result of the vote is displayed on the large display in form of six graphic bars, one for each voting option. The length of each bar corresponds to the proportion of votes each option received (Figure 11.3.). The results are displayed for 25 seconds and then a new voting interval starts.

The six options in the voting menu correspond to the six tracks in the multi-track music video. Only one track is visible at a given time, determined by the collective vote of the previous voting interval. The change of the video track results in a non-linear perception of the video on the public display.

The music video employed in the MobiLenin system comprises six tracks each showing a different performance style of the musician:

- clap: he claps hands to the rhythm of the music (no voice, only slim music version with no guitar sound and no singing);
- resign: no voice, just gestures, still slim music version with no guitar sound and no singing;
- guitar: he plays guitar (still no voice, reduced music version with guitar sound, but no singing);
- sing: he sings and plays guitar (full music version with guitar sound and singing);

- crazy: 'violent' performance (voice and full music version are on);
- skeleton: he turns into a skeleton (still playing guitar and singing with full music version on).

Although the performance style of the musician changes, the background footage stays the same.



Figure 11.2 Screenshots of the client's UI: (a) the voting interval has started; (b) casting a vote; (c) the vote is acknowledged; (d) a winning lottery coupon is received

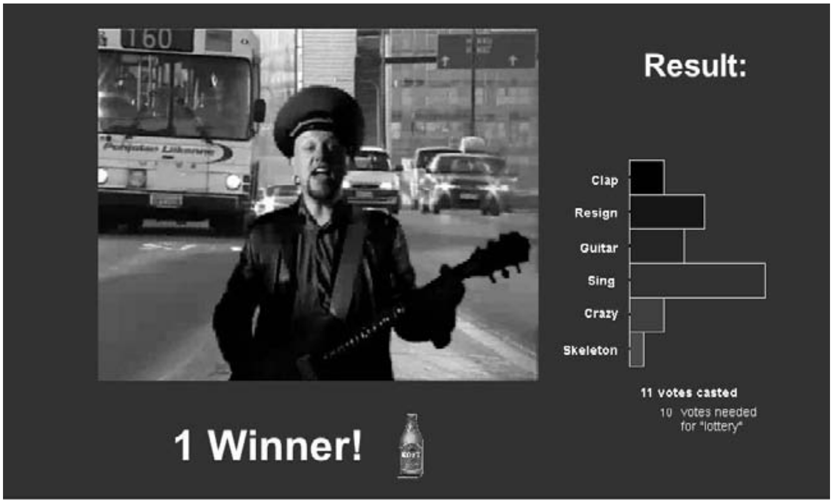


Figure 11.3 The public display

11.1.2 MobiLenin Mobile Client Code

Although you can't get this code to work since you are missing all the back-end applications, we want to give a rough explanation here to highlight how it was possible to rapidly prototype a project like MobiLenin with PyS60 in a matter of 2–3 days. The script is divided into two parts for better display (Examples 110 and 111). It might not contain the most elegant code, nevertheless it worked, served its purpose and shows you that you can program things in Python in many different ways. You are already familiar with some lines of code, for example the function `keys()` for handling keyboard keys.

The server side (external server) of the MobiLenin system consists of a few PHP scripts and some data files; the mobile client communicates with them over GPRS or 3G. At the startup of the application, a temporary id is requested from the external server by an HTTP request, `conn.request("POST", "/fetch_id.php")`. The `id` returned by the `fetch_id.php` script is used for further communication with the server. It stays valid as long as the mobile client is up and running.

The basic principle of the mobile client is that it polls the external server every two seconds to know whether to display the voting menu to the screen, fetch a winning coupon or simply remain waiting. The poll is done inside the `while` loop at the bottom of the script by a standard HTTP GET request, `conn.request("GET", "/control"+id+".txt")`, to fetch data from a file that resides on the external server. This data file holds the letters 'A', 'K', 'B' or 'P'. It is dynamically updated by the other back-end applications that control the voting cycle and run the video. With `r1 = conn.getresponse()` and `data1 = r1.read()`,

the received content from the data file is read into the variable named `data1`.

Let's look at the actions that follow based on the received content:

- 'A' triggers a pop-up note to the screen saying 'Voting is on' (Figure 11.2 (a)), then function `voting()` is called to display a pop-up menu with the voting choices (Figure 11.2 (b)) based on the list `choices=[u"Clap", u"Resign", u"Guitar", u"Sing", u"Crazy", u"Skeleton"]`. Resulting from the user's selection, a letter between 'A' and 'F' is sent to the external server to inform the overall vote count. This is done by a standard HTTP POST request, `conn.request("POST", "/voting_phone"+id+".php", params, headers)`. Then another pop-up note is triggered to the user saying 'Your vote is being processed' (Figure 11.2 (c)). Once this is done, the script keeps polling the server every 2 seconds.
- 'K' keeps the mobile client polling until a different letter comes in.
- 'B' (for beer) or 'P' (for pizza) notifies the user about being the winner of the lottery. A coupon (Figure 11.2 (d)) is fetched inside the function `winning()` from the server by the standard Python function `urllib.urlretrieve(url, tempfile)` and displayed to the screen. When the user presses the left softkey, the coupon disappears and the script keeps polling the server every 2 seconds.

Example 110: MobiLenin (1/2)

```
import httplib, urllib, appuifw, e32, graphics, key_codes

def keys(event):
    global win_state
    if event['keycode'] == key_codes.EKeyLeftSoftkey:
        win_state=0

def show_picture(picture):
    canvas.blit(picture)

def voting():
    choices=[u"Clap", u"Resign", u"Guitar", u"Sing", u"Crazy", \
                                                    u"Skeleton"]
    choice = appuifw.popup_menu(choices, u"Select + press OK:")
    choice_conversion={0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F'}
    params = urllib.urlencode({'data': choice_conversion[choice], \
                               'eggs': 0, 'bacon': 0})
    headers = {"Content-type": "application/x-www-form-urlencoded", \
               "Accept": "text/plain"}
    conn = httplib.HTTPConnection("www.yourdomain.com:80")
    conn.request("POST", "/voting_phone"+id+".php", params, headers)
    conn.close()

def winning(url):
    global win_state, coupon_shown
```

```
    if not win_state:
        if not coupon_shown:
            tempfile = "E:\\Python\\resources\\win.jpg"
            urllib.urlretrieve(url, tempfile)
            coupon_shown = 1
            win_state=1
            img_win=graphics.Image.open(tempfile)
            appuifw.note(u"Winner", "info")
            e32.ao_yield()
            show_picture(img_win)

appuifw.app.screen='full'
appuifw.app.body = canvas = appuifw.Canvas(event_callback=keys, \
                                           redraw_callback=show_picture)

img_wait=graphics.Image.open(u'E:\\Python\\resources\\wait.jpg')
keyboard_state={}
downs={}
running= 1
show_picture(img_wait)
voting_done = 0
win_state= 0
coupon_shown = 0
id = ""
```

Example 111: MobiLenin (part 2/2)

```
if id == "":
    conn = httplib.HTTPConnection("www. yourdomain.com:80")
    conn.request("POST", "/fetch_id.php")
    response = conn.getresponse()
    data = response.read()
    id = str(data)

while running:
    if not win_state:
        show_picture(img_wait)
        e32.ao_sleep(2.0)
        conn = httplib.HTTPConnection("www.yourdomain.com:80")
        conn.request("GET", "/ control"+id+".txt")
        r1 = conn.getresponse()
        data1 = r1.read()
        conn.close()

    if data1 == 'A':
        if not voting_done:
            coupon_shown = 0
            appuifw.note(u"Voting on", "info")
            voting()
            show_picture(img_wait)
            appuifw.note(u"Your vote is being processed!", "info")
            voting_done=1
            coupon_shown = 0
            show_picture(img_wait)

    elif data1 == 'K':
        voting_done=0
```

```
    elif data1 == 'B':
        winning("http://www.yourdomain.com/canvas_beer.jpg")
    elif data1 == 'P':
        winning("http://www.yourdomain.com/canvas_pizza.jpg")

    e32.ao_yield()
```

Example 112 lists the PHP script, `voting_phone"+id+".php`, which the mobile client calls when sending the vote to the external server. The PHP script receives the vote content in the parameter, `data`, and stores it in a file called `phone1.txt` which is read by the other back-end applications of the MobiLenin system. Each mobile client connected to the MobiLenin system has its corresponding data files and PHP scripts on the external server.

Example 112: MobiLenin server-side PHP script

```
<?php
$data = file_get_contents('php://input');
$filename = 'phone1.txt';
$handle = fopen($filename, 'a+');
fwrite($handle, $data);
fclose($handle);
?>
```

With this setup it is possible to run MobiLenin with many users at the same time as a multi-user entertainment game. A test with 75 simultaneous users was successfully carried out. For those interested in reading more about this project there is a 10-page research paper ([Scheible and Ojala 2005]), some documentation and videos at ***www.leninsgodson.com/mobilenin***.

11.2 Manhattan Story Mashup

Manhattan Story Mashup is an urban storytelling game, designed by the authors of this book, that combines mobile phones, the web and a large public display into interactive, collaborative street art. The game is based on real-time interaction between mobile phone and web users. A storytelling tool on the game's website allowed anybody to write stories that were illustrated in real-time by almost two hundred street players in New York, taking photos with Nokia N80 camera phones. Once a story was fully illustrated, it was presented on a large public display in Times Square (see Figure 11.4).

The street players were given points according to how many individual nouns – which were extracted from the web stories – they could illustrate successfully. The success was validated by another street player who



Figure 11.4 Manhattan Story Mashup in Times Square

was asked to match the newly taken photo with the original noun in a multiple-choice test. If she was able to match the correct noun with the photo, both the guesser and the illustrator were awarded points.

The game proved to be fun and engaging. During 90 minutes of playing, the street players took 3142 photos and made 4529 guesses to validate each other's photos. In total, 115 stories were written by the web players. While illustrating these stories, the street players visited 197 distinct GSM cells in midtown Manhattan.

A major factor of the success was a smooth and fast-paced user experience. The player had to think fast and act fast to find a suitable target in the urban environment which somehow represented a given noun. With one click, the user could accept a noun for illustration. After this, a countdown timer was shown which gave the player 90 seconds to shoot a suitable photo just by clicking the Select key once. The photo was then automatically uploaded to the server and the game client returned to show the list of available nouns. Thus, it was possible to play the game just by repeating this two-click, choose-and-shoot cycle.

Both the mobile client and the game server were implemented in Python. Starting from a vague game concept and no code at all, it took approximately three months to implement the client, the server and a highly dynamic website by one of the authors. The first prototype of the game client was finished in a weekend. The prototype was then field-tested and the second, improved prototype was implemented in two

weeks based on experiences from the first version. After this, another field-test was organized and the remaining rough edges were polished off to produce the final game client.

Without rapid prototyping and field-tests, it would have been impossible to achieve such a smooth user experience. Given only three months' development time, PyS60 was a perfect tool for the task. Being able to use the same language on both the client and the server effectively was a major benefit as well. As discussed in Section 10.3.3, the game client included a mechanism for automatic updating which ensured that last-minute bugs could not spoil the experience.

When implementing a system that orchestrates 200 mobile phone clients, a public web application and a large public display in real time, it is crucial that the implementation language does not increase complexity by imposing arbitrary constraints or conventions. With Python, all the pieces came together on the first try – something that anyone with tight schedules can appreciate.

An important success factor was that the client looked and felt like a game. Instead of using the standard user interface elements from the `appuifw` module, we implemented a custom-made set of UI elements, which were used to build the user interface for the game client (see Figure 11.5).



(a) (b) (c)

Figure 11.5 Manhattan Story Mashup user interface

There is nothing particularly difficult in making custom UI elements in PyS60. In our game client, the UI elements are drawn in the redraw function of the canvas, in a similar way to many examples in Chapter 5. The keyboard event handlers are used to modify some variables that affect how the element is drawn on the screen, for instance, which line on the list should be highlighted. This is similar to the UFO Zapper example in Section 5.5.

In Example 113, we show how the list element in Figure 11.5(a), is created. Although Example 113 is just an excerpt of a larger program, it should give you an idea of how to create custom UI elements.

Example 113: Manhattan Story Mashup custom list element

```
def list_redraw(self):
    self.img.rectangle(self.area, fill = WHITE)
    x = self.area[0] + 10
    y = self.area[1] + LIST_FONT[1]

    if len(self.list):
        sel_x = self.area[0]
        sel_y = self.area[1] + self.idx * (LIST_FONT[1] + 5)
        self.img.rectangle((sel_x, sel_y,\
            self.area[2], sel_y + LIST_FONT[1] + 4), fill = YELLOW)

    else:
        self.img.text((x, y), u"[empty list]", font = LIST_FONT,\
                                                     fill = BLUE)

    for item in self.list[self.first_vis: self.first_vis +
                                    self.nof_visible]:
        txt = u"%s" % self.filter(item)
        self.img.text((x, y), txt, font = LIST_FONT, fill = BLUE)
        y += LIST_FONT[1] + 5

    if self.first_vis + self.nof_visible < len(self.list):
        x = self.area[2] - 30
        y = self.area[3] - 30
        self.img.polygon((x, y, x + 20, y, x + 10, y + 10), fill = BLUE)

    if self.first_vis > 0:
        x = self.area[2] - 30
        y = self.area[1] + 20
        self.img.polygon((x, y, x + 20, y, x + 10, y - 10), fill = BLUE)
```

First, we clear the list area in white. Then, we position the coordinates x and y at the upper left corner of the list area. Depending on which item in the list is selected, a yellow rectangle is drawn to depict the highlighted element. If there are no elements in the list, the text '[empty list]' is shown.

After this, we loop through the visible list elements and draw them on the screen using the `Image.text()` function. During each iteration, the y coordinate is increased, so each item is drawn on a separate line.

If the list contains more items than those which are visible, a small triangle is drawn to notify the user that she can go up or down the list. One such triangle is visible in Figure 11.5(c).

As you can see, there is no magic in making custom UI elements. Although this approach requires much more work compared to using the standard UI elements from the `appuifw` module, it allows you to tailor the user interface to your particular application. If done well, this can

increase usability and lead to a smoother user experience. In any case, you are guaranteed to get a distinctive look for your program!

11.3 MobileArtBlog – Image-Composition Tool

The image-composition tool described in this section allows the user to compose and draw images, as seen in Figure 11.6. With the camera and navigation keys, a photo can be placed multiple times on the canvas and its size can be changed. Alternatively the photo can leave color traces by moving it on the canvas using the navigation keys. At any time during the composition process, a new photo can be taken. The

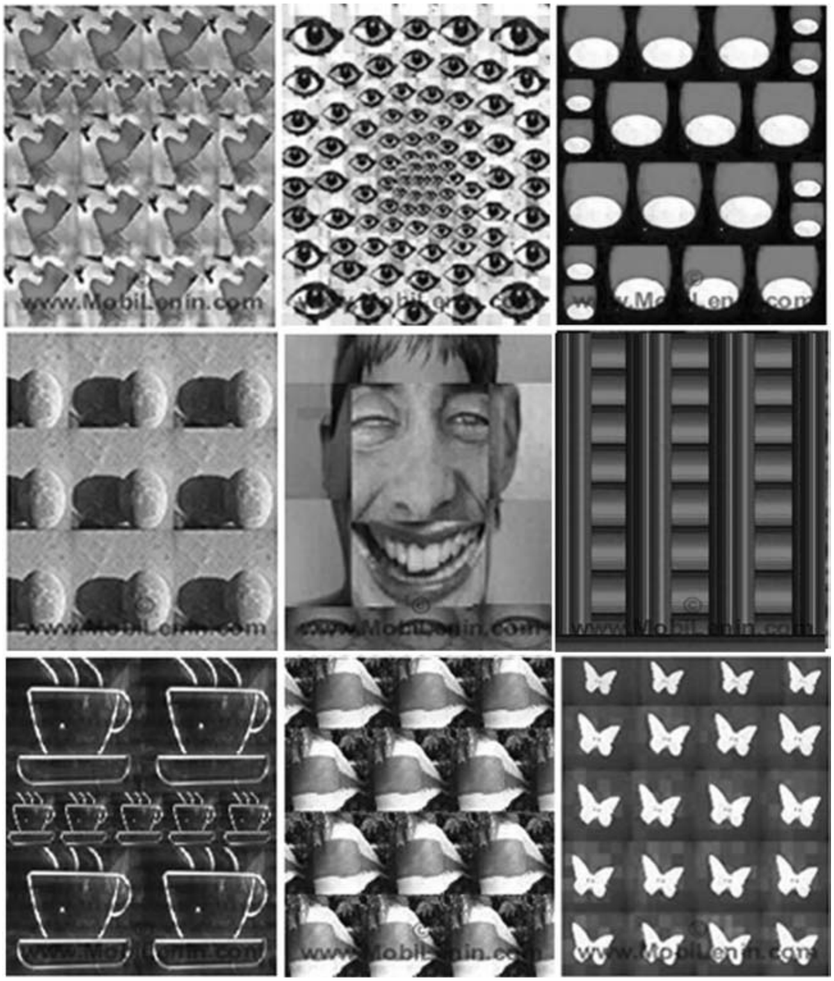


Figure 11.6 A collage of MobileArtBlog images

composed image is stored in the gallery of the phone and is also uploaded to the Internet by pressing only one button. The application was created by one of the authors for his MobileArtBlog concept. He uses it during his travels when he is stimulated by things he sees and experiences in different cities, places and situations, trying to capture the moment and turn it into a memorable 'art piece' (mobile art). Once the image is composed, it is posted directly over the mobile data network to his MobileArtBlog website with metadata such as the name of the place and its GPS coordinates. The GPS data are read by way of Bluetooth from an external GPS device. A large collection of created images can be seen at ***www.mobileartblog.org*** with their originating positions on a Google map view.

Let's look at the code to see how the application works. We list the mobile-client code, as well as the server-side PHP scripts that demonstrate how Python can easily be used to upload an image file to a URL. Finally we show how PyS60 can be used to insert data into a MySQL database.

11.3.1 MobileArt Client Code

Most of the code is similar to the examples in Chapter 5 regarding drawing graphics primitives and images and controlling graphic movements and the use of the camera. The MobileArtBlog is a combination of those applications with some additional functionality.

Example 114 presents the first part of the MobileArt client. It shows mainly the key-handling functions and some initialization logic. Example 115 presents most of the image-composing logic.

Example 114: MobileArtBlog (1/3)

```
import camera, appuifw, e32, graphics, key_codes

keyboard_state={}
downs={}

def handle_event(event):
    global downs,keyboard_state
    if event['type'] == appuifw.EEventKeyDown:
        code=event['scancode']
        if not key_is_down(code):
            downs[code]=downs.get(code,0)+1
        keyboard_state[code]=1
    elif event['type'] == appuifw.EEventKeyUp:
        keyboard_state[event['scancode']]=0

def key_pressed(scancode):
    global downs
    if downs.get(scancode,0):
        downs[scancode]-=1
```

```
        return True
    return False

def key_is_down(scancode):
    global keyboard_state
    return keyboard_state.get(scancode,0)

def quit():
    if appuifw.query(u"Save image","query") == True:
        background.save('e:\\Images\\art_picture.jpg')
    appuifw.app.set_exit()

def handle_redraw(rect):
    canvas.blit(background)

appuifw.app.body=canvas=appuifw.Canvas(event_callback = handle_event,\
                                     redraw_callback = handle_redraw)
appuifw.app.screen='full'
appuifw.app.exit_key_handler=quit
image_size = canvas.size
background = graphics.Image.new(image_size)
transfer_pic = graphics.Image.new(image_size)
transfer_pic.clear(0xffffff)
transfer_pic.save('e:\\transferpic.jpg')
picture = graphics.Image.new((50,50))
picture.clear(0x000000)
x = 100
y = 100
picture_size = 50
running = 1
drawing = 0
switch = 0
e32.ao_yield()
```

The basic concept of composing the art image is that a newly taken picture is stored as an `Image` object named `picture` and is drawn on the canvas at a certain position on top of the background image object, named `transfer_pic`. At any time, when the user presses the Select key, the background image object merges with the photo, so the photo is 'stuck' to the background, and is saved. The new background image object is then drawn to the canvas and the photo (in form of the `picture` image object) can be freely moved to the next position on the canvas using the navigation keys and so on. The size of the `picture` object can be changed with the 4 and 7 keys. The entire screen can be filled and the new art image composed.

Example 115: MobileArtBlog (2/3)

```
while running:
    if drawing == 0:
        background.clear(0xffffff)
        background.blit(transfer_pic,scale=1)
```

```
    background.blit(picture,target=(x,y,x+picture_size,
                               y+picture_size),scale=1)
    handle_redraw(())
    e32.ao_yield()

    if switch == 1:
        picture = camera.take_photo(size = (640,480))
    if key_is_down(key_codes.EScancodeLeftArrow): x -= 2.0
    elif key_is_down(key_codes.EScancodeRightArrow): x += 2.0
    elif key_is_down(key_codes.EScancodeDownArrow): y += 2.0
    elif key_is_down(key_codes.EScancodeUpArrow): y -= 2.0
    elif picture_size > 10:
        if key_is_down(key_codes.EScancode4): picture_size -= 2
    elif picture_size < 90:
        if key_is_down(key_codes.EScancode7): picture_size += 2
    elif key_pressed(key_codes.EScancodeLeftSoftkey): switch = 1
    elif key_pressed(key_codes.EScancode6): drawing = 1
    elif key_pressed(key_codes.EScancode9):
        background.save('e:\\transferpic.jpg')
        transfer_pic = graphics.Image.open('e:\\transferpic.jpg')
        drawing = 0
    elif key_pressed(key_codes.EScancodeSelect):
        background.save('e:\\transferpic.jpg')
        transfer_pic = graphics.Image.open('e:\\transferpic.jpg')
        switch = 0
    elif key_pressed(key_codes.EScancodeHash):
        background.save('e:\\transferpic.jpg')
        transfer_pic = graphics.Image.open('e:\\transferpic.jpg')
        upload()
```

To implement the photo leaving traces on the canvas, the background image object, `transfer_pic`, is prevented from being redrawn through setting the variable `drawing` to 0. When the user presses the hash button, the art image is uploaded to the MobileArtBlog website by the `upload()` function (Example 116).

Example 116 presents the third part of the MobileArt client, which deals mainly with the uploading of the art image to the MobileArtBlog website and its MySQL database by two HTTP POST requests:

```
conn1.request("POST", "/upload_to_url.php", chunk, headers)
conn2.request("POST", "/insert_artblog.php", params, headers)
```

In the first HTTP POST request, the art image – stored in the variable `chunk` – is handed over to the PHP script named `upload_to_url.php` (see Example 117), which saves the image to the server and returns its dynamically created filename as a URL back to the PyS60 script to the variable `picture_url`.

In the second HTTP POST request, some data such as location name, image URL and GPS data are first encoded and stored to the variable named `params` and are then pushed into a MySQL database by the PHP script named `insert_artblog.php` (see Example 118).

Example 116: MobileArtBlog (3/3)

```
upload():
    # text = u'name of location'
    # lat =  GPS data from external device
    # lon =  GPS data from external device
    f = open(u'e:\\transferpic.jpg',"rb")
    chunk = f.read()
    f.close()

    headers = {"Content-type": "application/octet-stream",
               "Accept": "text/plain"}
    conn1 = httplib.HTTPConnection("www.myserver.com")
    conn1.request("POST", "/upload_to_url.php", chunk, headers)
    response = conn.getresponse()
    picture_url = response.read()
    conn1.close()

    params = urllib.urlencode({'data': text , \
                               'eggs': picture_url, \
                               'bacon': lat, 'noodle': lon})
    headers = {"Content-type": \
               "application/x-www-form-urlencoded",\
               "Accept": "text/plain"}
    conn2 = httplib.HTTPConnection("www.myserver.com")
    conn2.request("POST", "/insert_artblog.php", params, headers)
    conn2.close()
    quit()
```

11.3.2 Server-side PHP Scripts

The server of the MobileArtBlog runs PHP and holds the PHP files in Examples 117 and 118. The first POST request from the phone passes the art image in binary form from the variable `chunk` to the script in Example 117, which stores it on the server in the directory named `pictures`. Its filename, which becomes part of the image's URL, is created dynamically using the timestamp and a random number. With `echo "".$filename;` the filename of the art image is returned to the PyS60 script.

Example 117: Server-side PHP script

```
<?php
// this file's name is upload_to_url.php
// read the incoming image data handed over from PyS60 phone

$chunk = file_get_contents('php://input');

// create a filename based om time and a random number
$timestamp = time();
$random_id = rand(0, 10);
$filename = 'Pic'. $timestamp .$random_id .'.jpg';
```

```
// write the file to the server into the directory pictures
$filepathname = "pictures/$filename";
$handle = fopen($filepathname, 'wb');
fputs($handle, $chunk, strlen($chunk));
fclose($handle);

// return the filename
echo " ".$filename;
?>
```

11.3.3 Inserting Data into a MySQL Database

The structure of the MySQL database table that is used for the MobileArt-Blog contains five fields that are named `blog_text`, `blog_datetime`, `blog_pic_url`, `blog_lon`, `blog_lat`. The second POST request passes the encoded contents sent by the PyS60 script to the PHP script in Example 118, which inserts them into the fields in the table.

Example 118: PHP script for MySQL database insert

```
<?php
// this file's name is insert_artblog.php
// Get the incoming params sent by the PyS60 phone

$data = $_POST['data'];
$eggs = $_POST['eggs'];
$bacon = $_POST['bacon'];
$noodle = $_POST['noodle'];

include "_mysql.php";

$sql = "INSERT INTO artblog (blog_text, blog_datetime,
        blog_pic_url, blog_lon, blog_lat)
        VALUES ( '$data', NOW(), '$eggs', '$bacon', '$noodle')";

db_query($insert, $sql);
?>
```

11.4 ArduinoBT Micro-Controller Board

As mentioned in Chapter 7, it is possible to connect your phone to a micro-controller. The ‘ArduinoBT board’ (Figure 11.7) is an example of a micro-controller board with Bluetooth extension chip that offers serial port communication.

Arduino is an open-source physical computing platform based on a simple I/O board and a development environment for writing Arduino applications (***www.arduino.cc***). The Arduino programming language is an implementation of Wiring (***http://wiring.org.co***), based on Processing (***www.processing.org***). It is easy to learn and quick to program. This makes it an ideal complement for PyS60 to do rapid prototyping of

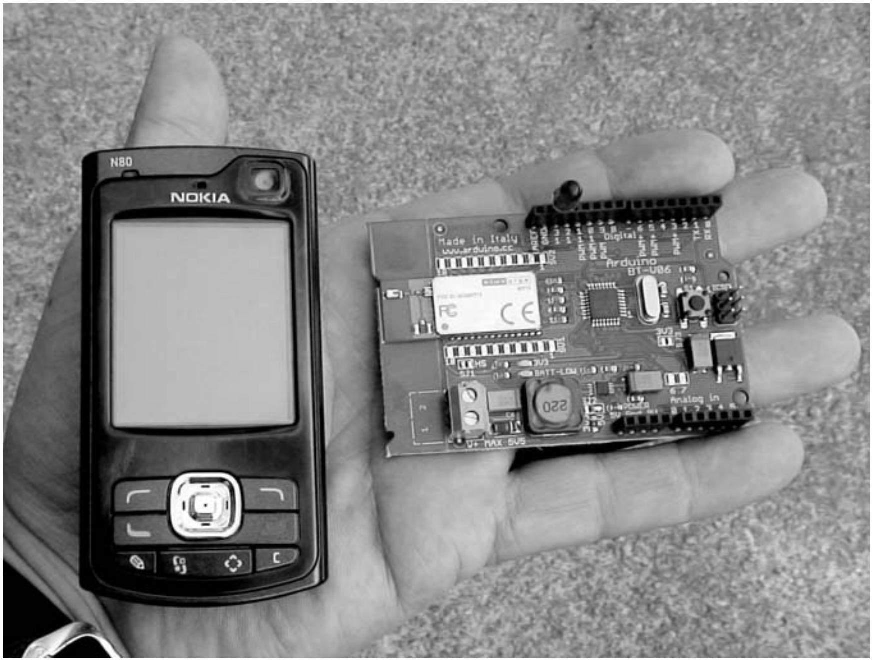


Figure 11.7 ArduinoBT board and Nokia N80

physical computing applications. It can serve as a mediating technology between sensors, motors and other actuators, providing access to the physical world.

The mobile phone acts as a gateway device to the Internet giving access to the digital and virtual world. As the Arduino board is small, light and battery-powered, it is suitable to be taken anywhere, like the mobile phone itself.

In this section, we describe the steps to connect a phone over Bluetooth to the ArduinoBT board. The PyS60 code is given, as well as the Arduino code that runs on the board. The Arduino software tool has a built-in editor for writing the Arduino code. When you press an upload button, the code is pushed to the board and can be executed.

The example application we provide here simply lets the user switch an LED light on and off on the Arduino board. Each time the LED changes its status, the board sends a confirmation (on or off) message back to the phone. This is a simple example but it shows you the basic principles for communicating with the board using the phone. It is up to you to do great things with it.

All you need for this example, besides the ArduinoBT board, is a battery between 1.2 V and 5 V for powering the board and a 5 mm LED that you stick into the board at pin 13 and pin GND.

11.4.1 Setting Up the ArduinoBT Environment

To set up the various components involved in this example, such as installing the Arduino software tool and configuring the Bluetooth settings on your computer, you need to take the following steps (this description was valid in June 2007 and is for a Mac, but similar steps apply for Windows PCs, too).

1. Create a Bluetooth serial port on your computer:

1. Go to System preferences. Select 'Bluetooth icon'.
2. Select 'Devices Tabs' and press 'Set Up New Device'.
3. Press 'Continue'.
4. For device type, select 'Any Device'.
5. ARDUINOBT should show up in a list, select it and press 'Continue'.
6. Type passkey '12345' (this is the default key set up by the factory).
7. Press 'Continue' (then you are done with the set up).
8. In 'Devices', you should now see ARDUINOBT (this is the default name set up by the factory, but you can change it if you wish).
9. Select the name ARDUINOBT.
10. Press 'Edit Serial Ports'. There you can see the name of your new port, for example, ARDUINOBT-bluetoothseri-1.

2. Set up the Arduino software

1. Download the Arbuino software from ***www.arduino.cc*** and install it on your computer.
2. Open it and select Tools.
3. Select 'microcontroller (mcu)' and set it as 'atmega168'.
4. Select Tools.
5. Select 'Serial port' and then the Bluetooth port that you created earlier, for example, /dev/tty.arduinobt-bluetoothseri-1.

Now you need to write your code for the board using the Arduino software. (There are plenty of tutorials available on how to do this, for example, at ***www.arduino.cc/***.)

3. Upload your Arduino code to the board

1. Press the Reset button on the Arduino board.

2. Press the ‘Upload to I/O board’ button on the Arduino software tool UI.
3. After a moment, the Arduino software tool should show ‘Upload done’.
4. Now you can start your Python script on the phone, connect over Bluetooth to the Arduino board and test your functionality.

11.4.2 Writing Code in the ArduinoBT Environment

In a similar way to Example 59 of Chapter 7, Example 119 first scans for Bluetooth devices until we find ARDUINOBT, the ‘nickname’ of the Arduino board and then selects it. This connects the phone to the Arduino board by the serial port using RFCOMM communication. The board can receive data from the mobile phone and send data back to the mobile phone.

The function `bt_send_data1()` sends the ASCII character ‘1’ to the board to switch the LED on. The function `bt_send_data2()` sends the ASCII character 0 to the board to switch the LED off. Each time the board receives a ‘1’ (49 in decimal format), it sends ‘1’ back to the phone which displays a note dialog ‘LED on’; similarly, it sends back ‘0’ (48 in decimal format) and the phone displays ‘LED off’.

Example 119: LED on/off

```
import socket, e32, appuifw

def choose_service(services):
    names = []
    channels = []
    for name, channel in services.items():
        names.append(name)
        channels.append(channel)
    index = appuifw.popup_menu(names, u"Choose service")
    return channels[index]

def connect():
    global sock
    address, services = socket.bt_discover()
    channel = choose_service(services)
    sock = socket.socket(socket.AF_BT, socket.SOCK_STREAM)
    sock.connect((address, channel))

def receive():
    global sock
    data = sock.recv(1)
    if data == "1":
        appuifw.note(u"LED on ", "info")
    elif data == "0":
        appuifw.note(u"LED off ", "info")

def bt_send_data1():
```

```
    global sock
    sock.send("1")
    receive()

def bt_send_data2():
    global sock
    sock.send("0")
    receive()

def exit_key_handler():
    print "socket closed"
    sock.close()
    app_lock.signal()

app_lock = e32.Ao_lock()

appuifw.app.menu = [(u"LED on", bt_send_data1),
                    (u"LED off", bt_send_data2),
                    (u"Connect", connect)]

appuifw.app.exit_key_handler = exit_key_handler
app_lock.wait()
```

Example 120 contains the code that runs on the Arduino board.

Example 120: Arduino code LED on/off

```
int LED = 13; // pin for LED
int RESET = 7; //reset pin for bluetooth
int val = 0; // initial serial port data

void ledOFF() {
  digitalWrite(LED, LOW);
}

void ledON() {
  digitalWrite(LED, HIGH);
}

void reset_bt(){
  // Reset the bluetooth interface
  digitalWrite(RESET, HIGH);
  delay(10);
  digitalWrite(RESET, LOW);
  delay(2000);
}

void setup() {
  reset_bt();
  pinMode(LED,OUTPUT);
  pinMode(RESET,OUTPUT);
  Serial.begin(115200);
}

void loop () {
  val = Serial.read();
  if (val != -1) {
```

```
    if (val == 49) {
      ledON();
      Serial.print(1); // feedback to mobile phone that LED is on
    } else if (val == 48) {
      ledOFF();
      Serial.print(0); // feedback to mobile phone that LED is off
    }
  }
}
```

11.5 Controlling Max/MSP with a Phone

A significant number of people in the art and design community use tools such as Pure Data (***www.puredata.org***), vvvv (***http://vvvv.org***) or Max/MSP or Jitter (***www.cycling74.com***) for interactive audiovisual applications and art installations. vvvv is especially good for real-time video synthesis that allows interaction with many users simultaneously.

These tools provide a graphical programming environment for music, audio and multimedia. All of them allow rapid programming of powerful audiovisual applications running on the computer or a server being controlled through multimodal interfaces. We want to show here how to use your mobile phone as a user interface for such tools. For practical reasons on our side, we have chosen Max/MSP to get the basic ideas across, but you can take the same approach when using one of the others as well.

First we describe how to use Bluetooth RFCOMM to set up the communication between PyS60 on the phone and Max/MSP on your computer that will allow you to switch a sound on and off and change its frequency. On the phone, we use a graphical switch and slider to manipulate the sound by sending relevant control data to Max/MSP.

We then describe a mobile multi-user scenario which allows multiple people to interact over WiFi instead, meaning that several users at the same time can control parameters of a sound generator using the same Max/MSP application.

11.5.1 Connecting a Phone to Max/MSP using Bluetooth RFCOMM

The patch of the Max/MSP graphical programming environment is shown in Figure 11.8.

An object with the text ‘serial a 9600’ handles the serial port communication in Max/MSP. When you start Max/MSP, this object opens one of the free serial ports of your computer. (You might have to check some additional parameters in your computer’s Bluetooth setting if it doesn’t work straight away.)

We cannot explain here the full functionality of the patch in detail, but the basic idea is that when the serial port receives the string ‘51’ over Bluetooth from the phone, the frequency of the sound produced by the

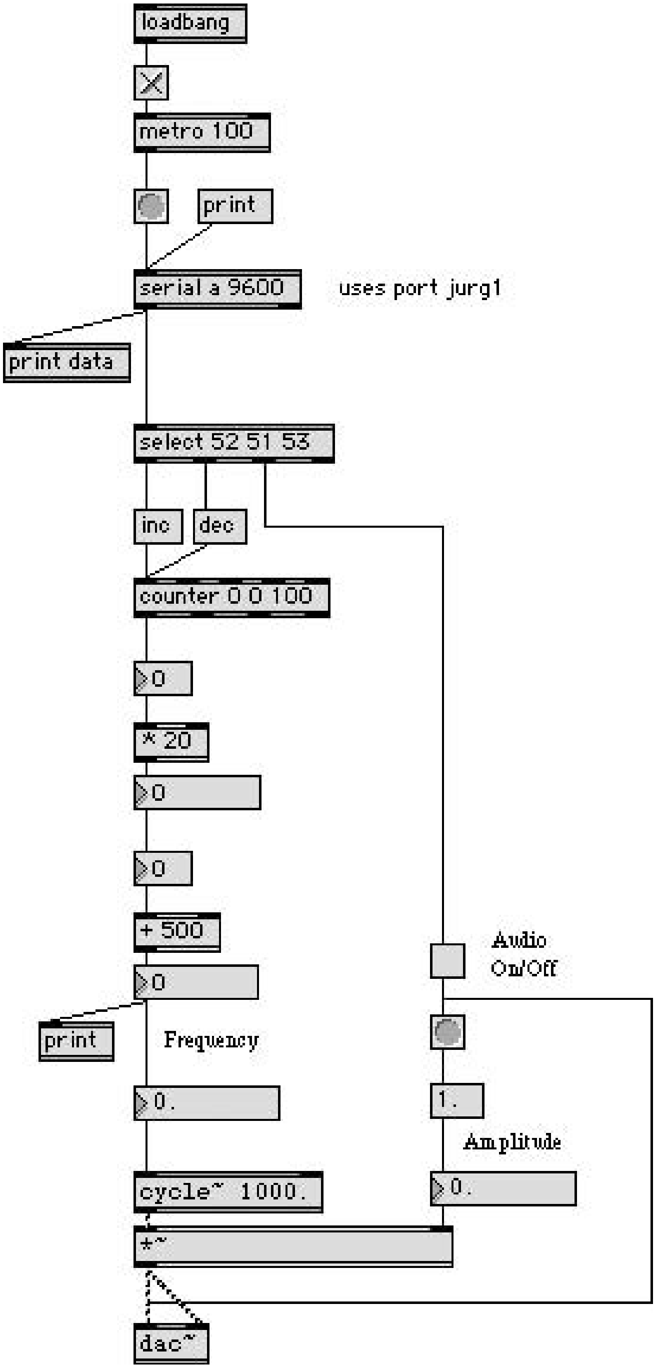


Figure 11.8 Max/MSP patch using Bluetooth

sound generator increases. Receiving the same string again increases the frequency. Receiving the string '52' decreases the frequency. When the patch starts up there is no sound to be heard, until the sound is switched on. This is done by sending the string '53' from the phone. To switch off the sound, the string '53' must be sent by the phone again.

In this case we are not using the built-in UI elements of PyS60, but instead we build our own graphical user interface (Figure 11.9) made up of JPEG images. We have a button for switching the sound on and off and a slider to change the frequency of the generated sound on the Max/MSP application. This interface uses six images (Figure 11.10).

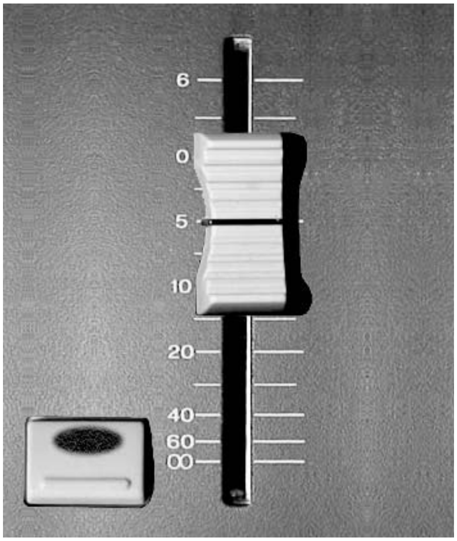


Figure 11.9 Graphical slider and switch for sound control

The code is divided into two parts (Examples 121 and 122). They need to be combined into one script to work. The first 20 lines of code are used to turn the graphical elements into ready-to-use image objects (see Chapter 5).

Whenever the screen must be redrawn (for example, when the position of the slider changes), we call the function `handle_redraw()` in which we use the concept of double buffering by copying all the graphical elements e.g. `img.blit(slidershaft, target = (0,0,w,h))` and `img.blit(contr, target=(142,y_pos_contr), mask=contrMask)` to the `img` object. We then blit the `img` object to the canvas:

```
canvas.blit(img, target = (0,0,w,h), scale = 1 )
```

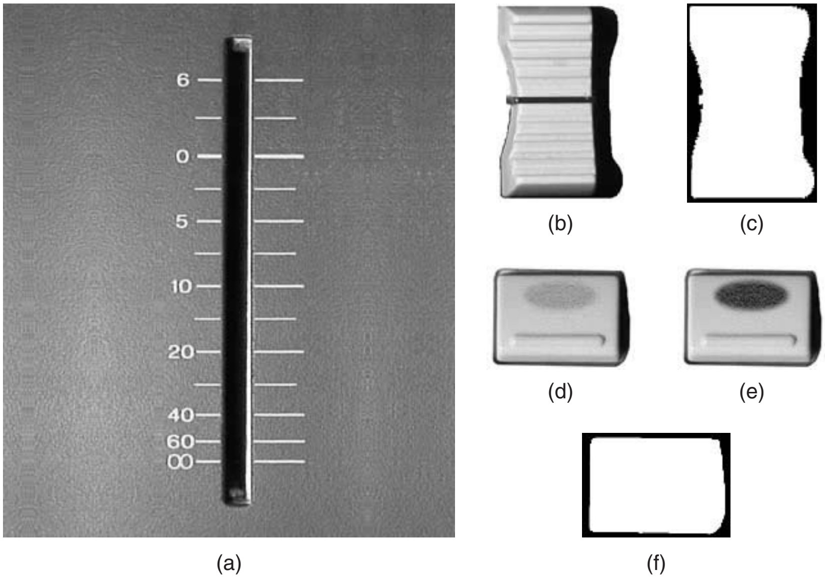


Figure 11.10 The images for slider and switch: (a) background, `background.jpg`, (b) slider, `controller.jpg`, (c) 1-bit mask for the slider, `controller_mask.png`, (d) switch on, `button_red.jpg`, (e) switch off, `button_dark.jpg`, (f) 1-bit mask for the switch, `button_mask.png`

To connect to Max/MSP over Bluetooth, we use the function `connect()` (Example 122), which includes the same code as in previous Bluetooth RFCOMM examples, basically setting up a socket and scanning for Bluetooth devices. For programming keyboard keys, we use the code that was described in detail in Section 5.2.2.

Example 121: Max/MSP using Bluetooth

```
import appuifw, e32, graphics, key_codes, socket

sound = 0
y_pos_contr = 100

slidershaft = graphics.Image.open("e:\\background.jpg")

makeMaskTemp = graphics.Image.open('e:\\controller_mask.jpg')
makeMaskTemp.save("e:\\controller_mask.png", bpp=1)
contrMask = graphics.Image.new(size = (97,149),mode = '1')
contrMask.load("e:\\controller_mask.png")
contr = graphics.Image.open("e:\\controller.jpg")

makeMaskTemp = graphics.Image.open('e:\\button_mask.jpg')
makeMaskTemp.save("e:\\button_mask.png", bpp=1)
buttnMask = graphics.Image.new(size = (111,78),mode = '1')
buttnMask.load("e:\\button_mask.png")
buttnOn = graphics.Image.open("e:\\button_red.jpg")
```

```
buttnOff = graphics.Image.open("e:\\button_dark.jpg")

def keys(event):
    global y_pos_contr, sound
    if event['keycode'] == key_codes.EKeyDownArrow:
        if y_pos_contr < 260 :
            y_pos_contr = y_pos_contr + 5
            sending(str(3))

    if event['keycode'] == key_codes.EKeyUpArrow:
        if y_pos_contr > 0 :
            y_pos_contr = y_pos_contr -  5
            sending(str(4))

    if event['keycode'] == key_codes.EKeySelect:
        if sound == 1:
            sound = 0
        else:
            sound = 1
        sending(str(5))

    handle_redraw(())
```

The 'Select' key is used to switch the sound on and off. Each time the 'Select' key is pressed we send the string '5' in ASCII format with `sending(str(5))` to Max/MSP. '5' is equal to 53 in decimal format and '53' is used in the Max/MSP patch.

The 'ArrowUp' key is used to increase the frequency of the sound generated by the Max/MSP application. When that key is pressed we send the string '3' in ASCII with `sending(str(3))` which is equal to '51' in decimal format used by Max/MSP. Also the y-position of the controller image is changed by 5 pixels with `y_pos_contr` = `y_pos_contr -5`, making it blit to the canvas slightly further up. The same logic is used for decreasing the frequency of the generated sound at the Max/MSP application with the 'ArrowDown' key instead (sending the string '4' in ASCII ('52' in decimal) as well as changing the y-position of the controller image downwards on the screen).

Example 122: Max/MSP using Bluetooth (2/2)

```
def handle_redraw(rect):
    global sound, img, w,h
    img.blit(slidershaft, target = (0,0,w,h))
    img.blit(contr, target=(142,y_pos_contr), mask=contrMask)
    if sound == 1:
        img.blit(buttnOn, target=(8,328), mask=buttnMask)
    else:
        img.blit(buttnOff, target=(8,328), mask=buttnMask)
    canvas.blit(img, target = (0,0,w,h), scale = 1 )

def choose_service(services):
    names = []
```

```
    channels = []
    for name, channel in services.items():
        names.append(name)
        channels.append(channel)
    index = appuifw.popup_menu(names, u"Choose service")
    return channels[index]

def connect():
    global sock
    address, services = socket.bt_discover()
    channel = choose_service(services)
    sock = socket.socket(socket.AF_BT, socket.SOCK_STREAM)
    sock.connect((address, channel))

def sending(data):
    global sock
    sock.send(data)

def quit():
    app_lock.signal()

canvas=appuifw.Canvas(event_callback=keys, redraw_callback=handle_redraw)
appuifw.app.body=canvas
appuifw.app.screen='full'
w, h = canvas.size
img = graphics.Image.new((w,h))
appuifw.app.exit_key_handler=quit
handle_redraw(())
connect()
app_lock = e32.Ao_lock()
app_lock.wait()
```

11.5.2 Connecting a Phone to Max/MSP using WiFi

When connecting a phone to Max/MSP using WiFi, we can use almost the same files as described in Section 11.5.1 about using Bluetooth. A few things are different and we explain the differences here.

We need to remove the `connect()` function and replace the code of the `sending()` function with the code in Example 123.

Example 123: Max/MSP using TCP/IP

```
def sending(data):
    HOST = '192.168.1.100' # The remote host
    PORT = 9000 # The same port as used by the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((HOST, PORT))
    s.send(data)
    print "data send:", data
    s.close()
```

We create a TCP/IP socket with `s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)` and use it to connect with `s.connect`

`((HOST, PORT))` to an IP socket on our computer identified by the IP address (HOST). We choose port 9000 for communication. Sending the control strings '51', '52' and '53' is done through `s.send(data)`.

In Figure 11.11, we can see the patch for Max/MSP. There is only one major difference between it and the Bluetooth diagram (Figure 11.8).

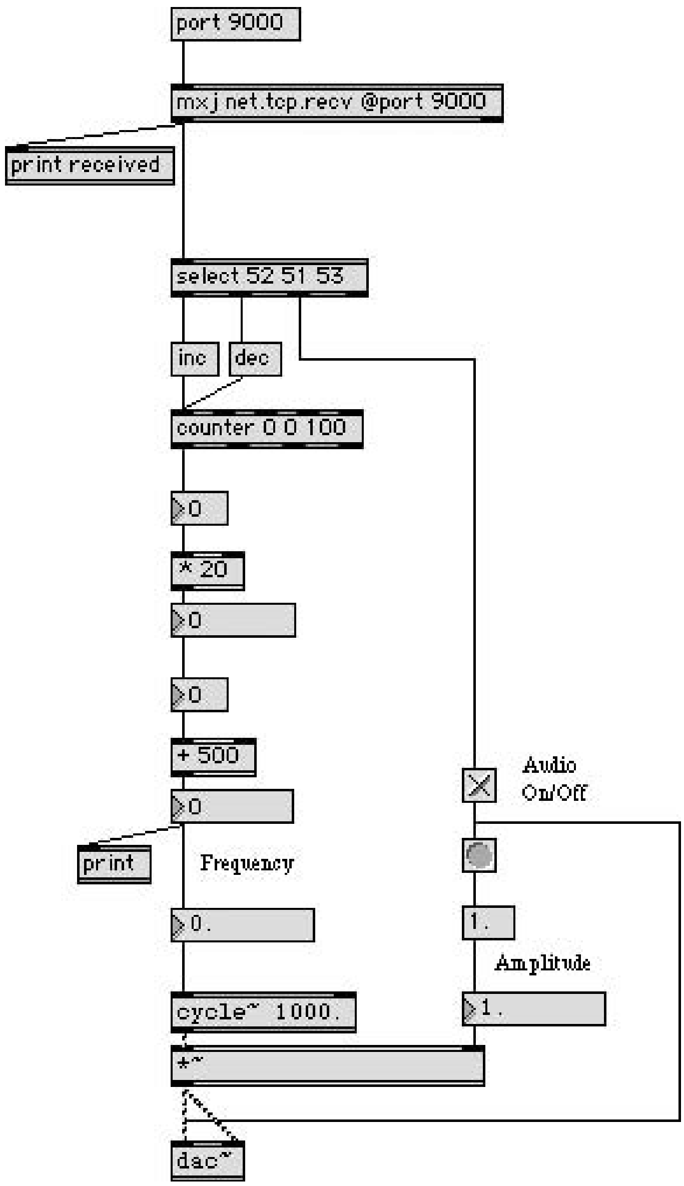


Figure 11.11 Max/MSP patch using TCP port

The TCP version uses the `mxj.net.tcp.rec@port 9000` object to receive the control strings, instead of the `serial a 9600` object. The `mxj.net.tcp.rec@port 9000` object handles socket communication over TCP in Max/MSP and is one of the simplest ways to handle HTTP transfers.

Once the Max/MSP application is running, the computer is ready to send and receive data over the socket. It is important that the communication is done through port 9000 using the current IP address of your computer. This means the port must be open and firewall settings must be set accordingly.

Environment B in Chapter 8 describes how to use your local computer to work as a test server which is accessed from Internet. But the scenario described here does not directly use Environment B. Instead, we want to set up a local WiFi network using a wireless broadband router to which we directly connect the computer running Max/MSP, as well as connecting all the mobile devices.

This scenario has many advantages. For example, you can run mobile multi-user applications over WiFi in any place without having to have a connection to the Internet, simply using your computer as a server for socket communication, to which the mobile devices connect. Further, this setup has the advantage that the wireless broadband router can be manually configured to assign an IP address to the computer. Therefore we know the IP address of our computer inside the local WiFi network and can use it to hardcode the destination address for 'server' socket communication which we need to specify in our PyS60 code (to successfully connect the mobile devices to the computer).

In the given scenario, each connected mobile device can switch the sound on and off as well as changing the frequency of the sound. This means that it can become messy when many users want to control one sound at the same time. There are many solutions to make the application more interesting and useful, such as having a controllable sound source with multiple parameters available for controlling different aspects of the one sound. Each mobile user can then control different parameters. But we want to leave it up to the readers of this book to expand the examples; this example just explains some basic principles.

Section 11.6 explains an important protocol for handling multi-user issues in the field of digital music: OpenSoundControl OSC.

11.6 OpenSound Control

OpenSound Control (OSC, ***www.cnmat.berkeley.edu/OpenSound Control***) is a protocol for communication among computers, sound synthesizers and other multimedia devices. It is optimized for modern networking technology. Often it is used for multi-user applications and

it is frequently deployed by the arts and design community. Interest in using mobile phones as part of multi-modal interfaces is on the rise.

We can use OSC with PyS60 by installing an additional Python library called `OSCmobile.py` to our phone into a folder named `python/Lib` on our memory card (if the folder doesn't exist, you can create it). You can download the library from ***www.mobilenin.com/pys60/oscmobile.htm***. This library is based on the OpenSound Control library implementation for Python by Daniel Holth and Clinton McChesney, found at ***http://wiretap.stetson.edu***.

In this example, we can use the Bluetooth code from Examples 121 and 122. All we need to change is to import the module `OSCmobile` and replace the code of the function `sending()` with the code in Example 124.

Example 124: OSC for mobile phones

```
import OSCmobile

def sending(data):
    global sock
    message = OSCmobile.OSCMessage()
    message.setAddress("/phone/user1")
    message.append(data)
    sock.send(message.getBinary())
```

This allows you to run exactly the same application setup as in Examples 121 and 122, except that you are now using OSC to encode messages before sending them to Max/MSP on the computer over Bluetooth.

In a multi-user scenario, different phones can use different names, for example user2 and user3, inside the address string of the OSC message to distinguish by which phone the message has been sent and to address, for instance, individual sound parameters.

Please check Chapter 7 to see how to receive data from a computer over Bluetooth RFCOMM. If your mobile phone receives OSC data, all you have to do is to use string handling (described in Chapter 4) to decode the OSC message and handle the OSC message contents further. To use OSC over TCP sockets for a mobile multi-user scenario, you could replace the code of function `sending()` in Example 123 with the code of Example 124.

11.7 Robotics

With a little creativity, a modern mobile phone can be used as a compact and capable robot brain. Besides having respectable computing power for an embedded device, the phone has the basic sensory functions built-in:

the camera can function as eyes and the microphone as ears for the robot. Because of its small size, the phone can easily be mounted on various platforms for locomotion, which may be controlled, for instance, over Bluetooth.

Nowadays, using cheap off-the-shelf parts, one can start experimenting on robotics without any soldering. This approach was taken by a course on Artificial Intelligence that was organized by the Department of Computer Science at the University of Helsinki in 2007. In this course, a Nokia N80 mobile phone, which served as an eye for the robot, was mounted on a Roomba robotic vacuum cleaner – the resulting chimera is shown in Figure 11.12. Since one can control Roomba remotely by sending commands to it over a Bluetooth serial port, it is possible to build a fully autonomous, visually guided robot based on these two widely available components.

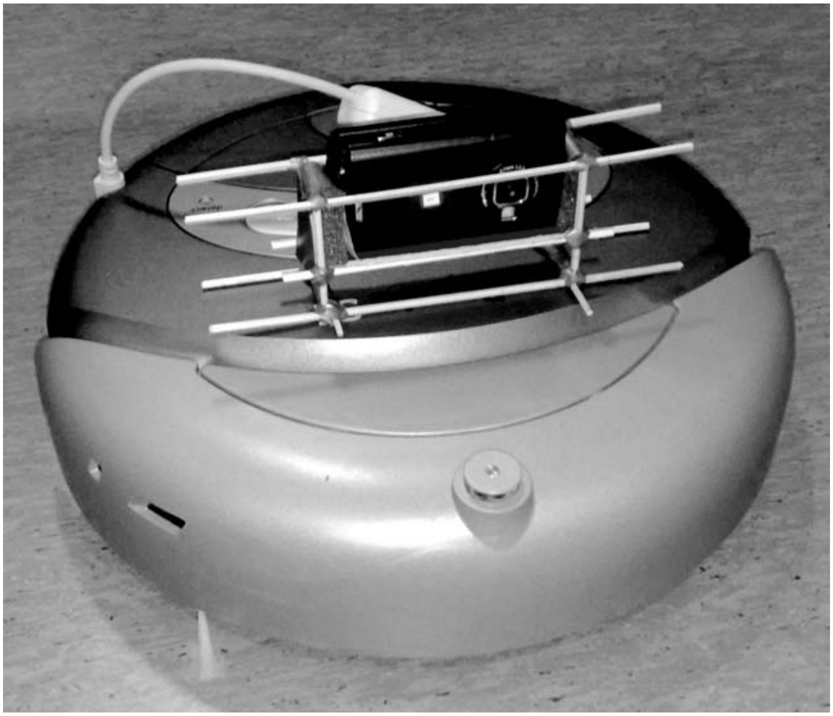


Figure 11.12 Nokia N80 mounted on a Roomba robotic vacuum cleaner

Students on the course were given a task to write a program that drives the robot through two gates, around a single pole and return through the gates back to the home base. To make the task easier for the students, the control program for the robot ran on a PC, which got a constant stream of photos from a PyS60 script running on the N80. Based on these visuals, the program was supposed to control the robot so that it stayed on track.

In further experiments, Roomba was controlled by the N80 and a PyS60 program in a totally autonomous manner. The phone controlled Roomba by Bluetooth using Roomba's proprietary, binary protocol for communication. The PyS60 program was able to take full control of the robot, as the protocol includes commands for driving the robot and collecting data from its sensors (see Examples 125 and 126).

Example 125: Roombatics (1/2)

```
import socket, time, struct

def bytecmd(cmd,*args): # args are bytes
    urn struct.pack("B%dB" % len(args), cmd, *args)

def intcmd(cmd,*args): # args are big endian ints
    return struct.pack(">B%dh" % len(args), cmd, *args)

print "OPENING BLUETOOTH SOCKET TO ROOTOOTH"
sock = socket.socket(socket.AF_BT, socket.SOCK_STREAM)
address, services = socket.bt_discover()
sock.connect((address,1))
roomba = sock.makefile("rw",0)

print "PUTTING ROOTOOTH INTO COMMAND MODE"
roomba.write("+++\r")
time.sleep(0.1)
if roomba.read(6) != "\r\nOK\r\n":
    raise Exception("+++ in failed")

print "ASKING ROOTOOTH TO TALK TO ROOMBA"
roomba.write("ATMD\r")
time.sleep(0.1)
if roomba.read(6) != "\r\nOK\r\n":
    raise Exception("ATMD failed")

print "SETTING ROOMBA TO SAFE MODE"
roomba.write(bytecmd(128))
time.sleep(0.1)
roomba.write(bytecmd(130))
time.sleep(0.1)
```

Example 126: Roombatics (2/2)

```
print "START DRIVING 250mm/s WITH RADIUS 2000mm LEFT"
roomba.write(intcmd(137,250,2000))
time.sleep(0.1)

print "GETTING SENSOR READING BYTES"
roomba.write(bytecmd(142,0))
time.sleep(0.1)
sensor_str = roomba.read(26)
sensor_bytes = ",".join(map(str,map(ord,sensor_str)))

time.sleep(1) # keep driving for an extra second
```

```
print "STOPPING -  DRIVING WITH VELOCITY 0"
roomba.write(intcmd(137,0,0))
time.sleep(0.1)

print "BACK TO ROOTOOTH COMMAND MODE"
roomba.write("+++\r")
time.sleep(0.1)
if roomba.read(6) != "\r\nOK\r\n":
    raise Exception("+++ back failed")

print "ROOTOOTH HANGUP"
roomba.write("ATDH\r")
time.sleep(0.1)
roomba.close()
print "THE END"
```

Examples 125 and 126 include a simple PyS60 script that drives Roomba forward, requests some sensor readings from it and stops. Besides showing what the Roomba control code looks like in practice, they also illustrate how one can use a binary protocol to communicate with a Bluetooth device. This is done using the module `struct`, which converts Python values to byte sequences according to the given format. You can find more information about this module in the Python Library Reference.

11.8 Summary

We hope that by going through these diverse real-world examples that are inspired by artistic approaches, we have motivated you to enable, release and nurture your creativity, so you can rapidly create mobile applications based on your own ideas and get ahead of most mobile users and even market trends.

You are experiencing technology needs today that will probably be experienced by many other users in the coming years. You know and understand well your own needs, local habits and traditions and they are close to the 'real situation'. This gives a big chance for you to come up with innovative ideas and develop products that will be appealing to others too. Put them out, share your innovations and see how happy you and other developers will be using your applications or developing them further.

PyS60 is a toolkit at your hand that can help you succeed. Surprise yourself and others with the applications you build.

Appendix A
Platform Security

A.1 Introduction

There are several releases of the S60 platform and they have some important differences. The most important division in the S60 platform is the division between S60 versions older than S60 3rd Edition (also known as S60 3.0) and versions after S60 3rd Edition.

Before S60 3rd Edition, a program written in native code was free to access any functionality available on the phone without asking confirmation from the user or being certified in any way. All programs were considered fully trusted. Once a program was running, it had the opportunity to do anything it wanted, including make the phone inoperable, give the user a large phone bill or spy on the user without their knowledge.

In S60 3rd Edition, the situation has changed with the introduction of a security framework known as Symbian OS Platform Security that limits what software running on the phone can do.

Platform Security is a complex topic and we cannot hope to cover it fully in this appendix. Instead, we will try to gather together in a unified form the parts that are most relevant to the Python for S60 programmer, with emphasis on independent prototype development and experimentation. Readers interested in more details are recommended to consult the documentation on the Symbian Signed website, ***www.symbiansigned.com*** and the definitive guide on the topic, [Heath 2006].

The policies followed by device manufacturers and Symbian in matters such as types of developer certificates granted and the actual security settings on the devices can vary across manufacturers, countries and device variants. The information on security settings in this appendix was correct at the time of writing, but the policies and processes may change.

From the viewpoint of the PyS60 programmer, the following limitations imposed by Platform Security are the most relevant:

- Accessing potentially sensitive features now requires capabilities.

- Certain files and directories are now considered protected and can be accessed only in a limited way. This is known as data caging.
- Installing new native applications to the device is only possible through the operating system's own software installer from a signed SIS file.

A.2 Capabilities

In a device that uses Platform Security, a program must have permission to access potentially sensitive features. In Platform Security jargon, these permissions are called capabilities. A program has to hold a certain capability to access a certain set of sensitive features. Not all features require capabilities – there are many things you can do without holding any capabilities at all. Almost all the examples in this book, except GSM locationing, do not require special capabilities.

There is a small, fixed set of capabilities and each capability grants access to a specific set of functionality. The capabilities defined in S60 3rd Edition are listed in Table A.1.

From the viewpoint of a PyS60 programmer, the capabilities can be roughly divided into three groups, based on the difficulty of accessing them:

- **User-grantable capabilities** are capabilities that the user who is installing a program can grant to the program at install time. A program that needs only user-grantable capabilities can be self-signed, meaning that it can be signed with a random untrusted key that anyone can generate.
- **Capabilities available with a free developer certificate (devcert)** are capabilities with which you can experiment on a single phone using a devcert available from the Symbian Signed service for free. However, packaging a program that needs these capabilities into a SIS file that would install to any phone can only be done through the Symbian Signed process.
- **Manufacturer-approved capabilities** are highly sensitive capabilities that can only be obtained from the device manufacturer, even if it's just for experimenting on your own phone. Getting these capabilities requires you to justify why you need them, and to have an ACS Publisher ID from Verisign.

The good news is that most of the things that you've learned to do in PyS60 so far need only user-grantable capabilities and none of them

Table A.1 Capability groups

Capability	Description
ReadUserData	Read access to user's confidential data such as contacts and messages
WriteUserData	Write access to user's confidential data such as contacts and messages
UserEnvironment	Access to confidential information about the user's environment by way of sensors such as microphone or camera
NetworkServices	Access to communications that may cost money such as telephone calls or SMS
LocalServices	Access to local communications that don't cost money to use, such as Bluetooth and infrared
Location	Location information, such as GPS coordinates
TrustedUi	Creating trusted user interface components
ProtServ	Registering server processes with a protected name
SwEvent	Generating simulated key events and capturing key events from any application
PowerMgmt	Killing processes or turning off the device
ReadDeviceData	Read-only access to sensitive system settings
WriteDeviceData	Write access to sensitive system settings such as system time, time zone and so on
SurroundingsDD	Access to device drivers that provide information about the surroundings of the phone
CommDD	Access to communication device drivers (for example, WLAN driver)
MultimediaDD	Access to multimedia device drivers
NetworkControl	Read or modify network protocol settings
DiskAdmin	Low-level disk administration functions such as formatting drives or mounting and unmounting partitions

(*continued overleaf*)

Table A.1 (*continued*)

Capability	Description
AllFiles	Full access to private directories; read-only access to `\sys`
DRM	Access to unprotected forms of DRM-protected content
TCB	Write access to `\sys` and `\resource` directories

Table A.2 Applications that need capabilities granted by a devcert

Description	Function or module	Required capabilities
Global key capture	keycapture module	SwEvent
Reading the cell ID	location.gsm_location()	ReadUserData, Location, ReadDeviceData
Reading the GPS location	position module	Location
Setting the system time	e32.set_home_time()	WriteDeviceData

need manufacturer-approved capabilities. There are a few cases that need devcert capabilities and they are listed in Table A.2.

If your program doesn't need any of these functions, or any third-party extension that would need extra capabilities, then you can avoid some of the effort related to code signing.

A.3 File System Protection

Besides the limitations described previously, access to some files is limited. There are three protected directories on each drive that have special properties. The protected directories are:

- `\sys`: Requires AllFiles to read, TCB to write.
  The most important part here is the `\sys\bin` directory since, under Platform Security, the phone can only execute native code in the `\sys\bin` directories. On previous S60 editions it was possible to store C++ extension modules for PyS60 in any directory, but in S60 3rd Eition they must all be stored in `\sys\bin`.
- `\resource`: Requires TCB to write and no capabilities to read.

This directory is used for storing shared, non-sensitive data that must be accessible to several programs. Python for S60 comes with a set of standard library modules written in Python and they are stored here.

- `\private`: Reading and writing to a program's own directory needs no capabilities. Reading other programs' directories requires AllFiles and writing to them requires TCB.

All other parts of the file system (for example, `\data\images`) can be freely read and written by all code. regardless of capabilities.

If you need to store program-specific data that must be easily accessible and writable by anyone, storing it in a program-specific subdirectory under `\data` is a good choice. The Nokia PC Suite file manager is able to access the entire contents of the memory card, but only the `C:\data` directory on the C: drive. The ability to use the file manager can be convenient especially during program development.

If you need to keep some or all of the data in your program private, you should store it in the directory `c:\private\<UID>`, where <\UID> is the UID code obtained from the function `appuifw.app.uid()`. Since only your program and programs that have the AllFiles capability are able to access this directory and AllFiles capability is granted to only very few specialized programs, your data is relatively safe if stored in `\private`.

A.4 SIS Package Signing

The only way to install native code to a normal device that uses Platform Security is through the Symbian OS software installer, from a signed SIS package. Before Platform Security, signing was optional but from Symbian OS 9.1 and the 3rd Edition of the S60 platform, it is mandatory if your program requires capabilities.

There are many different ways to obtain a signature for a SIS file and the way to use it depends mainly on the following factors:

- How and to whom you intend to distribute the SIS file? Is the number of target devices 1, 10, 100 or 100 000? How technically skillful are the intended users?
- What capabilities are needed for the code in that SIS file?
- Is the code intended for temporary or permanent use?

For a PyS60 programmer, the most interesting ways to obtain a signature are self-signing, signing with a devcert and going through the Symbian Signed process.

For the two former methods, you sign the SIS package yourself using a key and a certificate. In self-signing, you generate the key and certificate yourself and in signing with a devcert you obtain the certificate and key files by request from a network service hosted by Symbian.

The Symbian Signed process is different, in that the signing is not done by you. Instead, you submit your program for testing to a testing house, which sends you back a signed SIS package, provided your program passes the tests. See ***www.symbiansigned.com*** for more information.

A.4.1 Creating a Key and a Self-Signed Certificate

The tools for signing and creating keys and certificates are included in the S60 3rd Edition C++ SDK. Once you have installed the SDK, creating a self-signed certificate and the corresponding key takes just one command:

```
makekeys -cert -dname "CN=John Doe EM=john@doe.com" mykey.key mycert.cer
```

The – dname parameter can be used to identify the owner of the given key. Replace John Doe with your name and `john@doe.com` with your email address. There are also several other fields that you can add, such as:

- CO – two-letter country code of the country where you live, for example ''GB''
- OR – organization you are a part of, for example ''Acme Ltd.''

A.4.2 Getting a Developer Certificate

A developer certificate, devcert for short, is a key–certificate pair that can be used to sign SIS packages with higher capabilities than is available through self-signing, but with two important restrictions:

- The devcert expires at a specific time and after that SIS packages signed using the devcert can't be installed without tampering with the phone's clock.
- Packages signed with a devcert can only be installed into a specific, limited set of phones identified by their IMEIs. The number of devices that can be attached to a devcert depends on the type of devcert.

There are many different kinds of developer certificates that grant different levels of capabilities, enable installation to different numbers of devices and take different amounts of effort to obtain.

The capabilities granted by different devcerts and the specific procedures needed to obtain them are a matter of policy that may change in the

future. The reader is encouraged to check the Symbian Signed website for information regarding changes.

For the beginning Python for S60 programmer, the most interesting and practical type of developer certificate is available from a public Symbian network service. At the time of writing, this developer certificate can be used to install software onto one phone, with the user-grantable and free devcert capabilities.

Higher-grade devcerts are available and they may be used to get access to sensitive manufacturer-approved capabilities or to sign a SIS package for installation to more than just one device. The specifics of obtaining higher-grade devcerts is beyond the scope of this introductory appendix and the interested reader is encouraged to study the documentation available at the Symbian Signed website.

You can get a free devcert as follows:

1. Register into the Symbian Signed web service at ***www.symbiansigned.com.***
2. Download, install and run the Developer Certificate Request Tool.
3. Enter the IMEI of your phone and the capabilities you want. Typically you would want to select all the capabilities available with a free devcert.
4. The tool generates a private key and a certificate request file. Upload this into the Symbian Signed web service.
5. Download the finished developer certificate from the Symbian Signed service.
6. Proceed with signing and installation as described below.

Note: a devcert is only valid for a limited time. Remember to renew it before it expires! Also, make sure the clock on your phone is set correctly or the devcert will not work.

A.4.3 Signing the Python for S60 Interpreter with a devcert

In case you'd like to use free devcert capabilities with your Python interpreter software, all you need to do is to sign the Python ScriptShell file (e.g. `PythonScriptShell_1_4_0_3rdEd_unsigned_freedevcert.SIS`) with your devcert. The PythonForS60 file (e.g. `PythonForS60_1_4_0_3rdEd.sis`), which is the other installation file for your Python for S60 interpreter software, is already fully signed and doesn't require any action by you.

If you only want to use user-grantable capabilities there is no need to sign the installation file. You can use them as you downloaded them from the Sourceforge website.

A.4.4 Creating a Standalone Python Program

The Python for S60 SDK comes with a tool called `py2sis` that can be used to create SIS packages from PyS60 programs.

To use `py2sis`, you need:

- S60 C++ SDK, 3rd Edition, Maintenance Release
- Python for S60 SDK plugin
- Python 2.4 or greater

The tool is installed in the directory `\epoc32\tools\py2sis`. To turn a simple 'hello world' program into a SIS package perform the following steps:

1. Pick a UID for your program. A UID is a 32-bit number that identifies a given Symbian program and all programs installed in a device must have a unique UID. For use during development and testing you may pick a UID randomly from the range 0xe0000000–0xefffffff. If you intend to distribute your program, you should obtain a UID from the Symbian Signed website.
2. Invoke `py2sis` with a command such as the following (the UID is just for example – invent your own, don't use this one):

```
python \epoc32\tools\py2sis\py2sis.py hello.py hello.sis -sdk30
        -appname=HelloApp -uid=0xe0123456
```

3. A package called `hello.sis` is generated.
4. The package generated by py2sis is unsigned and must be signed before it can be installed. Signing the package with your key and certificate is accomplished by the following command:

```
signsis hello.sis hello-signed.sis mycert.cer mykey.key
```

where `mycert.cer` and `mykey.key` are the key and certificate to be used (either self-signed or devcert). Note that the key and certificate are given to the `makekeys` and `signsis` commands in the opposite order!

You are done! hello-signed.sis is now ready for installation.

A.5 Running Python for S60 Code under Platform Security

There are many different ways to run your PyS60 code in a device that uses Platform Security and you will typically use several different ways

during the lifetime of your program. The procedures needed are different for development time and deployment time: they depend on the set of capabilities needed and the type of the intended final audience.

A.5.1 Development

When developing your program, the typical procedure is to install the appropriate version of the PyS60 runtime package and the PyS60 interpreter to your device and test your program using that. The version of the PyS60 runtime needed depends on the capabilities you need:

- If your program requires only user-grantable capabilities, then this part is easy: you can just install and use the default version of the PyS60 runtime and script shell.
- If you need any devcert capabilities, then you will need to obtain a developer certificate for each device with which you want to test your code. Download the script shell package marked as 'unsigned-freedevcert', sign it with your devcert and install it.
- If you need manufacturer-approved capabilities, then you will need to obtain a manufacturer-approved devcert and recompile the PyS60 script shell (interpreter) to have these capabilities. The Python for S60 development team can provide advice in these cases.

A.5.2 Deployment

You can distribute your PyS60 programs in two main forms: as plain Python scripts or packaged into SIS packages signed in different ways. Both have their own advantages and disadvantages. The choice of deployment method depends on the capabilities you need and the type and size of your audience.

- Distributing as a plain script is simple for the developer and makes it easy for users to edit the scripts without extra tools. However, the users must run the program through the script shell, there is no way for the program to have its own icon in the application menu. Also, the capabilities available depend on the capabilities of the script shell; if the program requires more than user-grantable capabilities, people who wish to test it must obtain their own devcert. This method is suitable for only for development.
- Distributing as a self-signed SIS package is easy for the users to install and use – the program can have an application icon in the application menu. However, only user-grantable capabilities are available and warnings are displayed at installation time. This method also requires the developer to install additional tools. Some rare phone variants

do not allow the installation of self-signed packages. This method is convenient for both testing and deployment and can be used as long as the program doesn't require capabilities that aren't user-grantable.

- Distributing as an unsigned SIS package that testers sign with their own devcert allows the program to use user-grantable and free devcert capabilities and have an icon in the application menu. However, the users must obtain their own devcert and use the SDK signing tools to install the program. This method is suitable for testing with advanced users, but not for actual deployment.
- Distributing as a Symbian Signed SIS package means that all required capabilities are available and the program will install on any device. Some websites only accept Symbian Signed software for distribution. However, testing takes time and money (unless using the freeware signing process, which takes only time). This method is suitable for deployment of a finished program to the mass market.

Appendix B

Bluetooth Console

Bluetooth console makes it possible to use the PyS60 interactively from your PC. Instead of the typical cycle of 'edit, upload, test', you can evaluate lines of code in real-time on the phone.

This approach is particularly useful for experimenting with the standard modules. You can try out different functions and parameter combinations without having to edit or upload any files. Since the lines are executed on the actual device, you gain a realistic understanding of how long various functions take to execute and how different UI elements look on the screen.

It is also a rapid way to debug custom modules by you and other third-party developers. You can upload a new module to the phone, import it into the Bluetooth console and start testing its functions one by one. If exceptions occur, they are much easier to parse on the large display of your PC than on the phone.

If you are interested in some hard-core hacking, note that you can automate testing by the Bluetooth console since, after all, the console is just receiving lines of text from the PC. Instead of you typing the lines, they could be generated by, say, a Python script running on your computer. This might open up new possibilities for rapid prototyping and automated testing.

If interacting with the PyS60 shell programmatically from your PC sounds useful to you, have a look at the following files that are included in the PyS60 source distribution: `extras/examples/simplebtconsole.py` and `core/Lib/btconsole.py`. The former is a simplified implementation of the Bluetooth console and the latter the actual implementation of it. The PyS60 source distribution can be found at ***http://sourceforge.net/projects/pys60/***.

B.1 Setting up Serial Communication

To use the Bluetooth console, or communicate with the PC over Bluetooth from your own PyS60 applications, an RFCOMM serial port must be set

up on the PC side. On Mac OS X and Linux, the serial port is just a special file under the `/dev` directory. On Windows, a COM port is reserved for this purpose. In the following sections, we explain how to set up these interfaces.

First, set up the RFCOMM port as instructed in one of the sections B.1.1 to B.1.3. After you are finished with the setup, you can see if the serial port is visible to your phone as follows:

1. Copy Example 55 from Chapter 7 to your phone in the usual manner.
2. Run the script. You should see the name of your computer on the list of Bluetooth devices nearby. If you cannot see it, your PC cannot be discovered. Check any settings on your PC's Bluetooth configuration that might affect how other devices see your computer. Check also that Bluetooth is actually switched on.
3. Choose your computer from the list. The script should print out a list of services found on the PC. The list should include an entry such as 'Serial Port' (or 'PyBook', on OS X). If you can see a service like that on the list, the serial port is set up correctly. If not, the service is either not visible to this phone or it is not advertised correctly by the PC and you should re-check the settings.

B.1.1 Windows

The following steps should work for at least Windows 2000, 2003 and XP. Because of differences in Bluetooth drivers, in some cases the process may not be this straightforward and might require some further tweaking. In any case, the manual for either your computer or your Bluetooth dongle should explain how to set up the serial port for Bluetooth.

1. Open the Control Panel. Open Bluetooth Configuration.
2. Select the Accessibility tab and select the 'Let Other Bluetooth Devices Discover this Computer' option.
3. For the panel that specifies 'Devices allowed to connect to this computer', you can choose 'All devices' for testing. Note that this is insecure if you leave it on. Alternatively, you may select 'Add Device' and add your phone to the list.
4. Select the Local Services tab and select 'Add Serial Device'. Choose any available COM port and keep its number in mind. You will need it later.

B.1.2 Mac OS X

The following steps create a special file that can be used to communicate with the phone:

1. Open System Preferences dialog. Select Bluetooth.
2. Select the Settings tab and enable the 'Discoverable' box.
3. Select the Devices tab and, unless you have done this before, pair your Mac with your phone using the 'Set up new device' button. Select the Sharing tab, select Add Serial Port Service and name it 'PyBook' and for type, choose RS-232.

After you have gone through the above steps, you should see a dialog like the one in Figure B.1, which includes the new 'PyBook' item. As a result, a file at `/dev/tty.pybook` has been created that provides an RFCOMM interface.

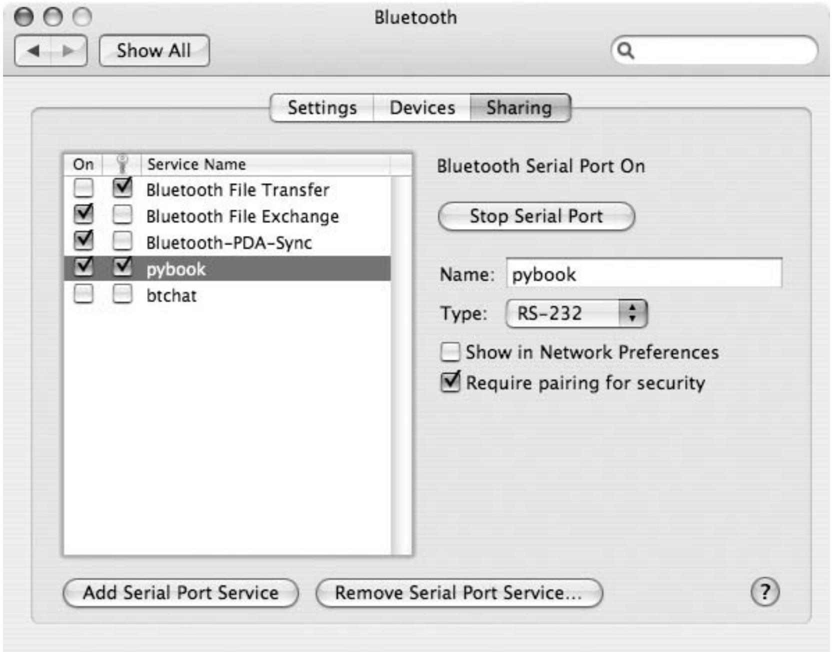


Figure B.1 RFCOMM setup on Mac OS X

B.1.3 Linux

Some Linux distributions may provide a graphical control panel that lets you configure the RFCOMM port, similarly to Mac OS X. However, here we perform the configuration on the command line, as this works on any distribution that has the Bluetooth tools (package `bluez-utils` in many distributions) installed.

Note that you should establish a Bluetooth pairing between your computer and the phone – see Section 7.1 for instructions. When you

have paired the devices successfully, you can execute the following commands as the root user:

- hciconfig hci0 piscan – sets the device discoverable
- sdptool add−channel=3 SP – advertises the available serial port on channel 3.
- rfcomm listen/dev/rfcomm0 3 – listens for incoming connections on channel 3. Note that this command does not return until you connect to the PC and close the connection on the phone side.

You should open another terminal in which you can run a server application of your own or the Bluetooth console. In this case, the active serial port can be found in the file `/dev/rfcomm0`.

B.2 Using Bluetooth Console

In this section, we assume that the RFCOMM serial port is set up correctly. You should also see the serial port service from the phone using Example 55, as described above. After this is done, there are only few reasons for the Bluetooth console not to work.

Bluetooth console is used in a terminal emulator application. Windows comes with one called HyperTerminal and several others are available for free. Linux and Mac OS X users have plenty of choices as well, but a program called `screen` is most often installed by default.

You can type any Python expression on the Bluetooth console. When the console seems to be active, you can try to type the following two lines on your PC and see what happens on the phone:

```
import appuifw
appuifw.note(u"Woohoo!", "info")
```

B.2.1 Windows

HyperTerminal can be found at Programs, Accessories, Communications, HyperTerminal. HyperTerminal is used as follows:

1. Open the program. First, it asks for a connection name, you can use for example 'btconsole'. Press OK and select the correct COM port. The port is the one you set up in the Local Services tab in the Bluetooth configuration panel. Select the fastest connection speed.
2. On your phone, open PyS60 interpreter and on the options menu, choose 'Bluetooth console'.

3. Your computer's name should show up in the list – choose it.
4. If the script executing on the phone prints 'OK', the console should be up and running. Hit enter on HyperTerminal to see the command prompt of the PyS60 interpreter.

If the above did not work, note that in some cases Nokia PC Suite may conflict with other users of the serial port. If you have PC Suite active, you might want to disable it for testing.

B.2.2 Mac OS X

After successful Bluetooth configuration, a serial port file should exist at `/dev/tty.pybook`. You can execute the following command on a terminal:

```
screen /dev/tty.pybook
```

Note that nothing is visible on `screen` until the console is running. Now you can run the Bluetooth console on your phone, as described in Windows steps 2 and 3. You should hit enter on the screen when the console is active on the phone side.

When you are finished with the console session, you can terminate the connection with Ctrl-D.

B.2.3 Linux

You should have an `rfcomm` process running on one terminal, saying 'Waiting for connection on channel 3'. Now you should open another terminal for the Bluetooth console and login as root to make sure that you can read and write/dev/rfcomm0. Then follow these steps:

1. Open the Bluetooth console on your phone, as described in Windows steps 2 and 3. If the connection was established successfully, `rfcomm` prints out a line such as 'Connection from 00:12:D2:DA:14:F4 to/dev/rfcomm0'.
2. Execute `screen/dev/rfcomm0` on the other terminal.
3. Hit enter to see the PyS60 command prompt on `screen`.

When you are finished with the console session, you can terminate the connection with Ctrl-D.

Appendix C

Debugging

Sometimes your Python program does not behave as you would expect. This happens in virtually every project at some point during the development process. It is a good idea to track down the cause as soon as you notice that something goes awry. This ensures that you always know that your program behaves as intended and it is built on a solid basis.

This appendix goes through the most typical ways of debugging a Python program. Since bugs are seldom really mysterious in Python, in contrast to many low-level languages such as C++, you can survive without heavy-duty debugging tools. Often a few well-placed print statements are enough to detect where the execution goes off track.

The ease of debugging in Python is based on rapid debug–evaluate iterations. Instead of assuming or trusting the documentation on how a PyS60 API function or a Python language construct behaves, you can try it in practice. Thus, the first step in successful debugging is to make sure that you can update your code on your phone with a minimal number of steps, as described in Chapter 2 and Section 10.3.

Be prepared to make lots of small changes to your code quickly. If modifying the code is a frustrating and time-consuming activity for you, check if you can streamline your development process. You should be able to update the program on your phone with two or three clicks from your PC.

Once you are fluent in updating the code, you should become accustomed to developing your code in short update–test cycles. If you make only a few small changes at a time and test and fix them immediately, you can be sure that your program works correctly as a whole. If it does not, at least you have a better understanding of how your program behaves and you can start applying methods from the debugging arsenal.

C.1 Interpreting Tracebacks

If execution of a Python program leads to an expression that is not understood by the interpreter, an exception is generated. Chapter 6

describes exceptions in detail. If the exception is not handled by your program, it is eventually shown on the Python console and the program execution is terminated.

Seeing an exception is useful in two respects: it tells you what and where something exceptional happened. Look at Example 127.

Example 127: See the error?

```
def hello():
    appuifw.query(u"Hello World", "txet")

hello()
```

You can probably spot the error. When the code is executed, the following output is printed on the console:

```
Traceback (most recent call last):
?
File "C:\python\ex_error.py", line 4, in ?
File "C:\python\ex_error.py", line 2, in hello
  def hello():
ValueError: unknown query type
```

We omitted some of the output between the first line and the last four lines. This output is called a *traceback*. It tells how the program execution proceeded just before the exception happened. The most recently executed lines are shown last. Almost always the last few lines pinpoint the actual location of the problem, but often it is useful to know how the program ended up on those lines, as shown by the full traceback. If you execute the program in the Python shell, the first lines of the traceback show how the program was started by the shell internally which is not really interesting to us.

In the example above, you can see that the exception occurred in line 2, in function `hello()`. The erroneous function was called by line 4 which does not belong to any function, thus it says 'in ?'. The files mentioned above these (which are not shown above) are not written by us, in contrast to `ex_error.py`, so we are not interested in them. When debugging programs of your own, you can follow the same principle: in the traceback, look for the lines which contain filenames familiar to you, starting from the bottom. This helps you to understand the context in which the exception occurred.

The last line of the traceback shows the actual exception, namely `ValueError` in the example above. The name of the exception may suggest the nature of the problem, but usually it is the error message which reveals the actual cause. Based on the traceback, we know that the error occurred on line 2, which contains a call to the function `appuifw.query()` which in turn generated the error message

'unknown query type'. Summing up this information, we notice that a typing error, `"txet"` instead of `"text"`, in the second parameter for the function call is the culprit.

Sometimes exceptions may be a bit misleading. Try out the code in Example 128.

Example 128: Syntax error

```
def hello():
    appuifw.query(u"Hello World", "text"

hello()
```

Once you run the code, you will get a traceback like this:

```
Traceback (most recent call last):
?
File "C:\python\ex_error.py", line 4
  hello()
SyntaxError: Invalid syntax
```

This suggests that the culprit is on line 4, but the function call `hello()` seems to be correct. In this case, the name of the exception, `SyntaxError`, reveals the actual cause.

The problem does not occur during execution of the program, but when PyS60 tries to read and interpret the code, which reveals any errors in the code text, that is, syntax errors. Since the exception is reported on line 4, which seems to be correct, we start to find the real culprit on the lines which were read before that. The true cause is a missing right parenthesis on line 2 which made the interpreter think that the function call continues all the way to line 4. The lesson here is that if you cannot spot the error on the reported line, start looking at previously interpreted or executed lines.

The most frustrating bugs are usually those which do not cause any exception but make the program behave in an unexpected manner. Typically, this happens when your program gets erroneous input which seems normal to the function but which makes it behave in a wrong way. To get rid of bugs like this, you need some other methods which are introduced in Section C.2.

In some cases, bugs may be hiding in innocent assumptions which just happen to be correct when you first run the program. For example, in the examples above a major bug is lurking. Both the examples are missing the `import appuifw` statement in the beginning of the script, which is needed to call `appuifw.query()`. The examples just happen to work since the Python shell imports the `appuifw` module internally. If you run the examples above as stand-alone programs, they would fail without any meaningful error message.

C.2 Debugging Procedure

How should you proceed when a program does not work as expected? After you notice that something is wrong, you have at least some information, namely, at which point the program does not work as expected. You should restart the program and repeat the steps that led to the problem to make sure that you have correctly recognized the context in which the bug occurs. You can repeat this many times if you like, maybe with slightly varying input to collect more information on the nature of the problem.

You should be able to locate the function which does not work correctly. Add some print statements in the code and try to find out what the erroneous values look like. Sometimes, examining the values is enough to reveal the actual cause. If you are debugging an application with a user interface, it might be easier to show the values in a pop-up dialog instead of using print statements and the console.

Once you have spotted the erroneous values, you should start backtracking to find out how they are produced in the first place. Usually, you do this by adding print statements along the program's path of execution, that is you add them to functions which were called before the function that manifests the problem.

Print out all intermediate values to check whether they match your assumptions as to what they should be. Once you have found the function which gets correct input but produces incorrect output, you are closer to identifying the actual cause of the problem. After you have found the cause, fixing it is often straightforward. Before you remove all print statements used for debugging, it makes sense to test the new function and ensure that it produces correct values. This makes the debugging of other errors easier.

If the incorrect function seems to be a PyS60 API call, that is, a function which is used to access the device's features, you should proceed as follows. First, double-check the documentation to make sure that you have understood the input and output of the API call correctly. Even though PyS60 documentation is typically written in a clear, technical, unambiguous way, sometimes the descriptions leave room for different interpretations. Try to find some example code using the particular API call in question, for instance from the web.

If you are sure that the function should work as you expect, make as minimal a test script as possible, which you can use to test the function in isolation. If you cannot get the function to work even in isolation, the problem is not likely to be in your program. Either the API function has bugs (not totally unlikely) or it is meant to be used in a different manner. In either case, you might ask for help in the Python for S60 mailing list, IRC channel, Forum Nokia discussion board or the PyS60 wiki.

If you feel adventurous, you could take a look at the Python source code and find out how the API call is implemented. This is not as intimidating as it may first sound, since the code is typically written in a clean and understandable style. Even if you do not understand it fully, it can point you in the right direction. Luckily Python is open source so this is a real option!

If the problem is not related to a device feature but to a Python language construct or to a function in its standard library, you can experiment with similar code on your PC. It is practical to have the standard Python interpreter, which is available at ***www.python.org***, installed on your PC – note that PyS60 is based on Python 2.2 and not the newest version, which is 2.5 as of July 2007. You can have a Python shell running on your PC on which you can try out different expressions and functions on the fly. This is often faster than sending the code to your phone to try out different expressions.

Appendix D

How to Use the Emulator

You download the PyS60 Interpreter installation files from ***http://sourceforge.net/projects/pys60***. You must download the correct version (for 2nd or 3rd Edition of S60). If you are not sure which one you need, check Table D.1 to find your phone model. You will also find a device overview at ***http://forum.nokia.com***.

Here are the steps to take to use the emulator:

1. Download and install the correct S60 Developer Platform Software Development Kit (SDK) which includes the emulator. The SDK can be found at Forum Nokia (***http://forum.nokia.com***).
2. Download and install the Python plug-in that comes as part of the appropriate SDK file, which can be found on the SourceForge PyS60 project website (***http://sourceforge.net/projects/pys60***).
3. Load your script into the emulator by copying your .py file to the appropriate folder (see the Python plug-in documentation).
4. Open the emulator and start the Python application using the icon on the emulator's desktop window or a subfolder of it.
5. Select Options, Run script. Choose your script from the list that appears and press OK. Your script should now start up.

Table D.1 PyS60 installation files

S60 Edition	PyS60 Install files
3rd Edition	
Nokia 3250, Nokia 5500 Nokia 5700, Nokia 6110 Nokia 6120, Nokia 6121 Nokia 6290, Nokia N71 Nokia N73, Nokia N75 Nokia N76, Nokia N77 Nokia N80, Nokia N91 Nokia N92, Nokia N93 Nokia N93i, Nokia N95 Nokia E61, Nokia E61i Nokia E65, Nokia E50 Nokia E60, Nokia E62 Nokia E70, Nokia E90	`PythonForS60_1_4_0_3rdEd.SIS` `PythonScriptShell_1_4_0_3rdEd_selfsigned.SIS`
2nd Edition	
Nokia 3230, Nokia 6600 Nokia 6260, Nokia 6620 Nokia 6670, Nokia 7610	`PythonForS60_1_4_0_2ndEd.SIS` `PythonScriptShell_1_4_0_2ndEd.SIS`
2nd Edition Feature Pack 2	
Nokia 6630, Nokia 6680 Nokia 6681, Nokia 6682	`PythonForS60_1_4_0_2ndEdFP2.SIS` `PythonScriptShell_1_4_0_2ndEdFP3.SIS`
2nd Edition Feature Pack 3	
Nokia N70, Nokia N72 Nokia N90	`PythonForS60_1_4_0_2ndEdFP3.SIS` `PythonScriptShell_1_4_0_2ndEdFP3.SIS`

References

Below, we have highlighted the most important sources of information in boldface.

Book References

Heath, C. (2006) *Symbian OS Platform Security: Software Development Using the Symbian OS Security Architecture*. Chichester: John Wiley & Sons

Lutz, M. (2001) *Python Pocket Reference*. O'Reilly

Lutz, M. and Ascher, D. (2003) *Learning Python*. O'Reilly

Scheible, J. and Ojala, T. (2005) 'MobiLenin – Combining a multi-track music video, personal mobile phones and a public display into multi-user interactive entertainment.' at ***www.leninsgodson.com/mobilenin***

Von Hippel, E. (2005) Democratizing Innovation. MIT Press

Other Book-Related Information

Website for this book: ***www.mobilepythonbook.com***

MobiLenin Python for S60 tutorials: ***www.mobilenin.com/pys60/menu.htm***

MobileArtBlog: ***www.mobileartblog.com***

Official Python for S60 page: ***http://opensource.nokia.com/projects/pythonfors60***

Python for S60 wiki: ***http://wiki.opensource.nokia.com/projects/Python_for_S60***

This is the best source for PyS60 material.

Forum Nokia Wiki: ***http://wiki.forum.nokia.com/index.php/Category:Python***

Python for S60 API documentation: ***http://sourceforge.net/projects/pys60***

The PyS60 documentation can be downloaded in PDF format.

Python for S60 IRC channel: #pys60 at freenode.net

Python for S60 discussion forum: ***http://discussion.forum.nokia.com/forum/forumdisplay.php?forumid=102***

Python Tutorials and Documentation

Peters, T. (2004) 'Zen of Python' at ***www.python.org/dev/peps/pep-0020***

Pilgrim, M. (2004) *Dive into Python*. Apress at ***www.diveintopython.org***

A free web book for experienced programmers

Scheible, J. (2007) 'Python for S60 tutorial' at ***www.mobilenin.com/pys60/menu.htm***

van Rossum, G. (2001) 'Python Style Guide' at ***www.python.org/dev/peps/pep-0008***

A comprehensive Python language lesson: ***http://docs.python.org/tut***

Official Python Library Reference: ***http://docs.python.org/lib***

This reference contains information about all standard modules that are not described in the PyS60 documentation. Note that only functions that are available for versions of Python before version 2.3 are available in PyS60 now.

Python Imaging Library (PIL): ***www.pythonware.com/products/pil***

Twisted: ***http://twistedmatrix.com***

A Python framework for building custom server software

Jabber: ***www.jabber.org*** and ***www.xmpp.org/rfcs*** for protocol documentation

Beautiful Soup: ***www.crummy.com/software/BeautifulSoup***

A Python HTML/XML parser

Glossary

Application	A program including a user interface that allows rich interaction with the user
Carrier	A mobile network operator
Cell phone	A mobile phone
Console	User interface of the Python for S60 script shell – either on the phone or on the PC over Bluetooth
Event	An external event, such as the user pressing a key or an SMS message arriving, that causes the program to react, typically by way of a callback function
Interpreter	A computer program that executes Python code; you have to install the Python for S60 interpreter on a phone to run Python programs
Library	Equivalent to a module or a collection of modules
MMS	Multimedia message service; an extension to SMS that allows sending of images and sounds or other rich types of data
Mobile network operator	A company that provides services for mobile phone subscribers
Mobile phone	A personal handheld communication device; modern mobile phones, especially smartphones, provide many functionalities, such as web browser, email, camera and music player, in addition to normal telephone functions

Module	A file that contains a collection of related functions and data grouped together; a program may import a module (unlike an object) only once and cannot handle several instances of the same module
Object	A basic building block of programs that contains both data (variables) and functions that are used to modify the data; practically everything in Python is an object, including strings, lists and functions
Program	A generic term for executable Python for S60 code; scripts and applications are types of program; modules are not programs, since they cannot be executed as such
PyS60	Abbreviation for Python for S60
Python	A high-level programming language which emphasizes the importance of programmer effort over computer effort and prioritizes readability over speed or expressiveness; comes with a large library of extension modules; Python programs are executed by the Python interpreter
Python for S60	The Python programming language on the S60 smartphone platform; includes the core Python language, many of Python's standard modules and a wide range of additional modules for accessing the phone's features, such as camera and networking
S60	A software platform for mobile phones based on the Symbian operating system; handles many high-level tasks, such as building user interfaces, on top of Symbian OS; all current high-end Nokia mobile phone models are based on the S60 platform.
Script	A small program that is typically used to automate a single task; performs little or no interaction with the user
Smartphone	A mobile phone that uses the S60 platform; more generally, a smartphone is a mobile phone with PC-like functionality
SMS	Short message service, often called text messaging; a way to send 160 characters of text from one mobile phone to another

Symbian OS	An operating system designed by Symbian for mobile devices; all current Nokia high-end mobile phone models are based on Symbian OS
Thread	A way to execute several tasks at the same time in a program; for example, one thread may listen to network events while another handles the user interaction
UI	User interface, made up of graphical elements, such as buttons, dialogs, menus and so on, that allow the user to affect program behavior
Unicode	An industry standard allowing computers to consistently represent and manipulate text expressed in any of the world's writing systems; in Python, Unicode strings are prefixed with the letter u, for example, `u'käärmekännykkä'`; in Python for S60, interfaces to the phone's features expect Unicode strings instead of normal strings

Examples

Python Language Lessons

These lessons teach basics of the Python programming language in a nutshell.

Python for S60 Modules

The following modules are used in this book. Custom modules and modules used on the PC side are not included in this list. More information about these modules can be found in the PyS60 API documentation and in the Python Library Reference.

Module Name	Description
appuifw	S60 user interface application framework; includes dialogs, notes, selection lists
audio	Recording and playing of audio files and text-to-speech engine
calendar	Calendar services: reading, creating entries, setting alarms
camera	Taking of photographs and starting and closing of the viewfinder
contacts	Address book services: finding and adding contact information
e32	Utilities related to Symbian OS that are not related to the user interface
e32db	Phone's internal relational database with a restricted SQL syntax

Module Name	Description
e32dbm	Phone's internal database with simple dictionary-like syntax
glcanvas	User interface control for displaying OpenGL 3D graphics
gles	Python bindings to OpenGL ES 3D graphics
graphics	2D graphics primitives and image loading, saving, resizing and transformation
httplib	Low-level access to HTTP and web
inbox	Reading of incoming SMS messages and deletion of SMS messages
key_codes	Identifiers for keyboard keys
keycapture	Global capturing of key events
location	GSM Cell ID location
md5	MD5 cryptographic hash function
messaging	Messaging services for sending SMS and MMS
os	Functions related to handling files and directories
os.path	Functions related to file names
position	Phone's internal GPS
random	Random number generator
socket	TCP/IP networking, Bluetooth, setting the default access point
sysinfo	System information of an S60 mobile device such as battery level, IMEI, signal strength or memory space
telephone	Telephone functionalities such as dial and hang-up

Module Name	Description
thread	Threads to handle concurrent processing of several tasks
time	Time and date functions
topwindow	Creating windows that are shown on top of other applications
urllib	High-level access to HTTP and web

Index